UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Decision Procedure for Synchronous Kleene Algebra

Luís Pedro Arrojado da Horta

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º Ciclo de Estudos)

Orientador: Prof. Doutor Simão Melo de Sousa

Covilhã, Outubro 2018

Dedication

Aos meus pais. À memória dos meus avós...

Acknowledgments

I would like to start by thanking Professor Dr. Simão Melo de Sousa, my supervisor, for the tremendous guidance, inspiration and availability provided, not only during the development of this thesis, but also throughout both my undergraduate and graduate courses.

I would also like to thank Professor Dr. Paul Andrew Crocker, for all the support and insightful advice given throughout my academic career.

A special thanks to my parents for all the emotional and financial support provided throughout all my personal and academic life.

Another special thanks to Filipa Gameiro, my girlfriend, for all the support and the enormous encouragement you have given me, and of course for spell checking this document.

As for my friends I would like to start by thanking João Reis, for all the fruitful conversations, steering me in the right direction when I was feeling lost and for your friendship!

I would also like to thank my friends for all the support during my academic life, specially, Rui Paiva and Eunice Martins. It is a blast working with you guys!

Finally, I would like to thank *RELiablE And SEcure computation group* (RELEASE) and all its members, for providing me with all the necessary equipment, allowing me to develop this and many other projects.

Resumo

A Kleene Algebra (KA) é um sistema algébrico que tem bastantes aplicações quer no campo da matemática como também da informática.

Foi batizada com o nome do seu inventor *Stephen Cole Kleene*, que ao longo da sua carreira fez um estudo intensivo sobre expressões regulares e autómatos finitos [Kle56].

Quando há necessidade de raciocinar equacionalmente sobre programas, recorre-se frequentemente à Kleene Algebra, visto que esta consegue exprimir noções de escolha, composição sequencial e até a noção de iteração. A necessidade de raciocinar equacionalmente sobre ações que podem ser executadas de forma concorrente levou ao aparecimento da Algebra de Kleene Síncrona ou Synchronous Kleene Algebra (SKA). Esta última foi introduzida por *Cristian Prisacariu* em 2010 no seu artigo [Pri10] como uma extensão à Kleene Algebra mas que contém uma noção de ação concorrente.

A equivalência de linguagens é um problema perene em ciências da computação. Nesta dissertação iremos apresentar ao leitor uma explicação detalhada de um processo de decisão para termos de Synchronous Kleene Algebra (SKA) bem como a sua implementação utilizando a linguagem de programação `OCaml`.

Palavras-chave

Álgebra de Kleene, Álgebra de Kleene Síncrona, Processo de Decisão

Resumo Alargado

A Kleene Algebra (KA) é um sistema algébrico que tem bastantes aplicações quer no campo da matemática como também da informática.

Foi batizada com o nome do seu inventor *Stephen Cole Kleene*, que ao longo da sua carreira fez um estudo intensivo sobre expressões regulares e autómatos finitos [Kle56].

Quando há necessidade de raciocinar equacionalmente sobre programas, recorre-se frequentemente à Kleene Algebra, visto que esta consegue exprimir noções de escolha, composição sequencial e até a noção de iteração. A necessidade de raciocinar equacionalmente sobre ações que podem ser executadas de forma concorrente levou ao aparecimento da Algebra de Kleene Síncrona ou Synchronous Kleene Algebra (SKA). Esta última foi introduzida por *Cristian Prisacariu* em 2010 no seu artigo [Pri10] como uma extensão à Kleene Algebra mas que contém uma noção de ação concorrente.

A equivalência de linguagens é um problema perene em ciências da computação. Nesta dissertação iremos apresentar ao leitor uma explicação detalhada de um processo de decisão para termos de SKA bem como a sua implementação utilizando a linguagem de programação `OCaml`.

Este processo de decisão é a conjunção de dois trabalhos. O primeiro desenvolvido por *Broda, S., et al.* no artigo [BCFM15] onde através de derivadas parciais sobre expressões regulares construímos um autómato que aceita a linguagem gerada por essa mesma expressão regular. O segundo desenvolvido por *Filippo Bonchi* e *Damien Pous* no seu artigo [BP11] onde propõem uma otimização ao algoritmo de *Hopcroft* e *Karp* [HK71] baseada em *bi-simulações até à congruência* para comparação entre autómatos.

Abstract

Kleene Algebra (KA) is an algebraic system that has many applications both in mathematics and computer science. It was named after Stephen Cole Kleene who extensively studied regular expressions and finite automata [Kle56].

Moreover it is often used to reason about programs, as it can represent sequential composition, choice and finite iteration. Furthermore, the need to reason about actions which can be executed concurrently, spawned SKA. SKA is an extension of KA introduced by *Cristian Prisacariu* in [Pri10] that adopts a notion of concurrent actions.

Laguange equivalence is an imperishable problem in computer science. In this thesis we present the reader with a detailed explanation of a decision procedure for SKA terms and an OCaml implementation of said procedure as well.

Keywords

Kleene Algebra, Synchronous Kleene Algebra, Decision Procedure

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

iff if and only if

KA Kleene Algebra

KAT Kleene Algebra with Tests

SKA Synchronous Kleene Algebra

EOL End Of Line

EOF End Of File

NFA Nondeterministic Finite Automaton

DFA Deterministic Finite Automaton

Chapter 1

Introduction

1.1 Context

A Kleene Algebra (KA) is an algebra that can modulate regular events. It arises both in mathematics and computer science. For instance, *Backhouse, R., et al.* said in [BKM01], that Kleene Algebra has been successfully applied to give a semantic description of imperative programs with non-deterministic choice. Additionally, *Cristian Prisacariu* said in [Pri10] that they can be viewed as a formalism that can be used to reason about programs. Furthermore Kleene Algebra has been used in graph paths [BMR98] and in networking [AFG+14, FKM+15].

The need to reason about actions which can be executed concurrently, spawned Synchronous Kleene Algebra (SKA). SKA is an extension of KA introduced by *Cristian Prisacariu* in [Pri10] that adopts a notion of concurrent actions. We hope to entice the reader to further pursue this amazing field that is Kleene Algebra.

1.2 Problem Statement

In computer science, the problem of language equivalence arises in many areas, such as Language Theory or Compiler Theory. Deciding regular expression equivalence or Kleene Algebra term equivalence can be viewed as part of a set of problems that is Language Equivalence. Furthermore one can say that Kleene Algebra is often referred to as the Algebra of Regular Expressions, since it presents an easy way to express patterns.

In this document we will focus on deciding Synchronous Kleene Algebra SKA. Moreover, we will try to assertively answer the question: *"given two SKA terms, α, β, is $\alpha \sim \beta$?"* where $x \sim y$ means that x is language equivalent to y.

1.3 Contributions

In this thesis we present a detailed explanation of a decision procedure for SKA. This procedure is the combination of the work of *Broda, S., et al.* in [BCFM15] and the work of *Bonchi, F., et al.* in [BP11]. The former allows us to construct an automaton based on the partial derivatives of a given SKA term (denoted on the sequel by $\mathcal{T}_{SKA}$) and the latter provides us a method based on *bisimulation up-to congruence* for automata comparison. Moreover, we provide an implementation of said procedure using the `OCaml` programming language.

1.4 Structure of the thesis

For a better comprehension of this thesis, this document is structured in the following way:

1. First chapter – **Introduction** – Contains a brief introduction to the project, some background information one needs to know in order to fully understand it, the problem that it aims to solve, and also the structure of the document.

2. Second chapter – **Introduction to Kleene Algebra** – Defines the notation used in the document, it also introduces the reader to Kleene Algebra by explaining the formalisms involved as well as some applications of KA.

3. Third chapter – **A Decision Procedure for Synchronous Kleene Algebra** – Describes thoroughly a decision procedure for $\mathcal{T}_{SKA}$. Additionally this chapter contains a toy example of the aforementioned procedure.

4. Fourth chapter – **Implementation and Validation** – provides the reader with some of the most important details of implementation of the decision procedure described in chapter 4. Moreover it shows some tests that were conducted in order to validate such implementation.

5. Fifth chapter – **Conclusion and Future Work** – summarises this project and provides the reader with some of the most important conclusions drawn from this thesis. Finally, a discussion of future work is given.

Chapter 2

Introduction to Kleene Algebra

2.1 Introduction

This chapter will provide a background to Kleene Algebra and will also explain why KA and SKA are promising tools in computer science. Moreover it also gives the reader some background definitions which will be used in the remainder of this document.

This chapter is organised into the following sections:

- section 2.2 – **Definitions and Notations** – Presents the reader with some concepts and terms used throughout the document.
- Section 2.3 – **Kleene Algebra and Synchronous Kleene Algebra** – Describes the formalisms used in KA and SKA
- Section 2.4 – **Why use Kleene Algebra** – Explains the benefits of using KA. Furthermore it describes known KA and SKA applications.
- Section 2.5 – **Related Work** – Describes some of the related work done on deciding KA.
- section 2.6 – **Conclusions** – Contains the most important conclusions gathered from this chapter.

2.2 Definitions and Notations

There are several definitions of the following algebraic structures in both mathematics' and computer science's literature. In this document we chose to use the one given by *Dexter Kozen* in his lectures [Koz04]. The next sections will show said definitions.

2.2.1 Semigroups, Monoids and Semirings

We shall start with some formal definitions before we proceed, in particular to define what is a semigroup, as well as a monoid.

A *semigroup* is an algebraic structure denoted $(\mathcal{S},\cdot)$ where $\mathcal{S}$ is a set, $\cdot$ is an associative binary operation on $\mathcal{S}$ such that $\forall x,y,z \in \mathcal{S} \rightarrow x\cdot(y\cdot z) = (x\cdot y)\cdot z$.

A *monoid* is an algebraic structure denoted $(\mathcal{M},\cdot,1)$ where $(\mathcal{M},\cdot)$ is a semigroup, 1 is a distinguished element of $\mathcal{M}$ that is both a left and right identity for $\cdot$, such that $\forall x \in \mathcal{M}, \rightarrow 1\cdot x = x\cdot 1 = x$. For a better understanding of monoids some examples are given below:

- $(\Sigma^*,\cdot,\varepsilon)$ where Σ^* is the set of finite length strings over an alphabet Σ, $\cdot$ is concatenation of strings, and ε is the null string.
- $(2^{\Sigma^*},\cup,\emptyset)$ where 2^{Σ^*} is the powerset or set of all subsets of Σ^*, $\cup$ is set union, and $\emptyset$ is the empty set.
- $(\mathbb{N},+,0)$ where $\mathbb{N}$ is the set of natural numbers.
- $(\mathbb{R}_+ \cup \{\infty\}, min, \infty)$ where $\mathbb{R}_+$ denotes the set of non-negative real numbers, ∞ is a special infinite element greater than all real numbers, and *min* gives the minimum of two elements.
- $(R^{n\times n},\cdot,I)$ where $R^{n\times n}$ denotes the set of $n\times n$ matrices over a ring R, $\cdot$ is ordinary matrix multiplication and I is the identity matrix.
- $(X \rightarrow X, \circ, \iota)$ where $X \rightarrow X$ denotes the set of all functions from a set X to itself, $\circ$ is function composition and ι is the identity function.

What is a semiring? A *semiring* is an algebraic structure denoted $(\mathcal{S},+,\cdot,0,1)$ such that $(\mathcal{S},+,0)$ is a commutative monoid, $(\mathcal{S},\cdot,1)$ is a monoid, $\cdot$ distributes over $+$ on both left and right sides, 0 is an annihilator for $\cdot$ in the sense that $\forall x \in \mathcal{S}, 0\cdot x = x\cdot 0 = 0$. A semiring is idempotent if and only if (iff) $\forall x \in \mathcal{S}, x+x=x$.

We can now define an *idempotent semiring* to be any structure $(\mathcal{S},+,\cdot,0,1)$ satisfying the following identities for all $x,y,z \in S$:

$$\begin{array}{ll}
x+(y+z)=(x+y)+z & x+y=y+x \\
x+0=x & x+x=x \\
x(yz)=(xy)z & 1x=x1=x \\
x(y+z)=xy+xz & (x+y)z=xz+yz \\
0x=x0=0 &
\end{array}$$

2.2.2 Order

As human beings, we tend to assign a certain order to everyday objects. Such order can be interpreted as a relation between objects. In certain sets, it is imperative that we can establish an order as well. A partial order on a set is a binary relation that is reflexive $(\forall x, x \leq x)$, antisymmetric $(\forall x,y,\ if\ x\leq y \wedge y\leq x \implies x=y)$ and transitive $(\forall x,y,z,\ if\ x\leq y\wedge y\leq z \implies x\leq z)$. Any idempotent semiring has a naturally-defined partial order $\leq$ associated with it, such that

$$x \leq y \overset{def}{\iff} x + y = y \quad (2.1)$$

The proof that $\leq$ is a partial order is shown in [Koz04] and goes as follows: reflexively can be proven by the idempotence axiom $(x + x = x)$, anti-symmetry follows from commutativity of the operator $+$, $(if\ x \leq y \wedge y \leq x \implies y = x + y = y + x = x)$. And finally for transitivity if $x \leq y$ and $y \leq z$ then

$$\begin{array}{rcll} x + z & = & x + (y + z) & \text{since } y + z = z \\ & = & (x + y) + z & \text{associativity of } + \\ & = & y + z & \text{since } x + y = y \\ & = & z & \end{array}$$

Therefore we can conclude that $x \leq z$ and thus finalising the proof.

2.2.3 Automata

In this subsection we will present the reader with definitions for both Deterministic Finite Automaton (DFA), and Nondeterministic Finite Automaton (NFA). We will start with the former.

Definition 1. *DFA. A Deterministic Finite Automaton (DFA) is a tuple $(S, \Sigma, \delta, I, F)$ where:*

- *S is a finite set of states;*
- *Σ is a finite set of input symbols, also known as Alphabet;*
- *$\delta : \Sigma \times S \to S$ is the transition function;*
- *I is the initial state $(I \subset S)$;*
- *F is the set of final states $(F \subseteq S)$.*

We now present a more general type of finite automaton, NFA, which has two main differences regarding DFA. The first one is the fact that there might be more than one transition associated with the same symbol of the alphabet from the same state. The second one relies on the possibility of performing a transition between states, without any symbol of the alphabet being associated with said transition. Furthermore, this type of "symbolless" transitions are usually refered to as ε-transitions.

Definition 2. *NFA. A Nondeterministic Finite Automaton NFA is a tuple $(S, \Sigma, \delta, I, F)$ where:*

- *S is a finite set of states;*
- *Σ is a finite set of input symbols, also known as Alphabet;*

- $\delta : (\Sigma \cup \varepsilon) \times S \to \mathcal{P}(S)$ *is the transition function;*
- I *is the initial state* $(I \subset S)$*;*
- F *is the set of final states* $(F \subseteq S)$.

Please note that in the NFA case, $\delta_\sigma(x)$ outputs the set of states that are valid transitions from state x associated with the symbol $\sigma \in \Sigma$. Therefore, the output of this function is in the powerset of S. Moreover we require δ to be a total function. We consider $\delta_\sigma(x) = \emptyset$ if there are no transitions from state x associated with the symbol $\sigma \in \Sigma$.

2.2.4 Notation

This subsection contains some notations that will be used throughout the remainder of this document. Unless explicitly said otherwise, the reader should refer to this subsection for the meaning of some operators.

- $+ \to$ is the choice operator (usually referred to as ”or” in programming).
- $\cdot \to$ is the concatenation operator or the sequence operator.
- $\times \to$ is the synchrony operator (more details in 2.3.2).
- $\cup \to$ is the regular union.
- $\mathcal{L}(\alpha) \to$ denotes the language generated by α.
- $\partial_\sigma(\alpha)$ denotes the set of partial derivatives of a term α with respect to the letter σ.
- $\alpha \sim \beta$ denote that α is language equivalent to β (i.e. $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$).

2.3 Kleene Algebra and Synchronous Kleene Algebra

We hereby present the most important section of this chapter. Kleene Algebra KA was named after *Stephen Cole Kleene* who extensively studied regular expressions and finite automata [Kle56]. It was further studied by *Conway* in [Con12] and *Dexter Kozen*, some of Kozen’s work can be found on [Koz94, Koz97, Koz03]. In this section we offer the reader some insight not only into KA and SKA but also to Kleene Algebra with Tests (KAT).

2.3.1 Kleene Algebra

There are several definitions of KA, and hereby we present a definition proposed by *Broda, S., et al.* in [BCFM15], that goes as follows:

A Kleene Algebra can be defined as a structure $(\mathcal{K}, +, \cdot, ^*, 0, 1)$, where * is a postfix unary operation on $\mathcal{K}$, 0 and 1 belong to $\mathcal{K}$, such that $(\mathcal{K}, +, \cdot, 0, 1)$ is an idempotent semiring satisfying properties (Axioms) 2.2 to 2.5, as well as all the identities shown in subsection 2.2.1. The natural order $\leq$ in $(\mathcal{K}, +, \cdot, 0, 1)$ is defined by $x \leq y$ iff $x + y = y$.

$$1 + xx^* \leq x^* \quad (2.2)$$

$$b + ax \leq x \rightarrow a^*b \leq x \quad (2.4)$$

$$1 + x^*x \leq x^* \quad (2.3)$$

$$b + xa \leq x \rightarrow ba^* \leq x \quad (2.5)$$

The axioms for the * operator are equations 2.2 and 2.3, and two implications or *Horn Formulas* [Hor51] 2.4 and 2.5. The significance of axioms 2.2 to 2.5 concerns the solution of linear inequalities, as well as axioms 2.2 and 2.4 provide the least solution to a certain single linear inequality in a single variable. An example of such solution is the fact that said axioms together say that a^*b is a solution to 2.6.

$$b + aX \leq X \quad (2.6)$$

One can easily verify the veracity of the previous claim simply by using axiom 2.2. If one takes into account the monotonicity of multiplication and distributivity $1 + aa^* \leq a^*$ becomes $(1 + aa^*)b \leq a^*b$ which in turn becomes $b + a(a^*b) \leq a^*b$.

Taking into account the information described above we can now talk about Kleene Algebra with Tests (KAT). Informally, Kleene Algebra with Tests (KAT) is just a Kleene Algebra KA combined with a Boolean Algebra in order to accommodate tests [Pri10]. These tests are Boolean values of 0 and 1. Moreover one can formally define a Kleene Algebra with Tests to be variant of KA as defined by *Dexter Kozen* in [Koz97], and the definition goes as follows: A KAT is a two-sorted algebra $(\mathcal{K}, \mathcal{B}, +, \cdot, ^*, 0, 1, ^-)$ where $\mathcal{B} \subseteq \mathcal{K}$, and $^-$ is a unary operator defined only in $\mathcal{B}$ such that $(\mathcal{K}, +, \cdot, ^*, 0, 1)$ is a Kleene Algebra and $(\mathcal{B}, +, \cdot, ^-, 0, 1)$ is a Boolean Algebra.

2.3.2 Synchronous Kleene Algebra

Synchronous Kleene Algebra was introduced by *Cristian Prisacariu* in 2010 [Pri10] and it combines Kleene Algebra with a synchrony model of concurrency from *Robin Milner's* SCCS calculus [Mil83]. One of the reasons that inspired *Prisacariu*'s research was the need to represent and to reason about actions that can be performed simultaneously.

As said by *Prisacariu* in [Pri10], SKA has three particularities. The first one is that it formalises a notion of concurrent actions based on a synchronous model. The second one being the fact that *actions* are interpreted as sets of synchronous strings. These same actions can also be represented as a special type of finite automata which accepts the same sets of synchronous strings that form the models of the actions. And finally, the third particularity is that SKA incorporates a notion of conflicting actions.

The formal definition of a SKA is a structure $(\mathcal{K}, +, \cdot, \times, ^*, 0, 1, \mathcal{K}_b)$, where $\mathcal{K}_b$ is a finite set of atomic actions, $(\mathcal{K}, +, \cdot, ^*, 0, 1)$ is a Kleene Algebra (KA), and $\times$ is a new operator which represents the synchronous composition of two actions. This new binary operator is associative, commutative, distributive over $+$ with absorvent element 0 and identity 1. Furthermore, it respects axioms 2.7 - 2.14. Regarding axiom 2.14, it is referred to as the *synchrony axiom*, where $\mathcal{K}_b^\times$ is the smallest subset of $\mathcal{K}$ that contains $\mathcal{K}_b$ and is closed for $\times$. The operator precedence goes as follows, $+ < \cdot < \times < {}^*$. The set of languages over an alphabet $\Sigma = \mathcal{P}(\mathcal{K}_b)\backslash\{\emptyset\}$ is the standard model of SKA over $\mathcal{K}_b$. These types of languages are called *synchronous languages*. Every $a \in \Sigma$ denotes a synchronous concurrent action.

$$a \times (b \times c) = (a \times b) \times c \quad (2.7)$$

$$a \times b = b \times a \quad (2.8)$$

$$a \times 1 = 1 \times a = a \quad (2.9)$$

$$a \times 0 = 0 \times a = 0 \quad (2.10)$$

$$a \times a = a, \forall_a \in \mathcal{K}_b \quad (2.11)$$

$$a \times (b + c) = a \times b + a \times c \quad (2.12)$$

$$(a + b) \times c = a \times c + b \times c \quad (2.13)$$

$$(a_\times \cdot a) \times (b_\times \cdot b) = (a_\times \times b_\times) \cdot (a \times b), \forall_{a_\times, b_\times} \in \mathcal{K}_b^\times \quad (2.14)$$

We are now able to define the operations of "$\times$". As described in [BCFM15], the synchronous product of two letters in Σ is simply their union. The synchronous product of two words $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_n$ where $n \geq m$ is given by $x \times y = y \times x = (a_1 \cup b_1 \cdots a_m \cup b_m) b_{m+1} \cdots b_n$.

Finally the synchronous product of two languages L_1, L_2 is given by

$$L_1 \times L_2 = \{x \times y \mid x \in L_1, y \in L_2\}$$

One can inductively define the language $\mathcal{L}(a)$ as follows:

- $\mathcal{L}(a) = \{\{a\}\}$
- $\mathcal{L}(0) = \emptyset$
- $\mathcal{L}(1) = \{\varepsilon\}$
- $\mathcal{L}(a^*) = \mathcal{L}(a)^*$
- $\mathcal{L}(a + b) = \mathcal{L}(a) \cup \mathcal{L}(b)$
- $\mathcal{L}(ab) = \mathcal{L}(a)\mathcal{L}(b)$
- $\mathcal{L}(a \times b) = \mathcal{L}(a) \times \mathcal{L}(b)$

2.4 Why use Kleene Algebra

Kleene Algebra is widely used in mathematics and computer science. For example, it is used in relational algebras [AU79] [Pra90] as well as in Propositional Dynamic Logic [Pel85] [HTK00]. Also, *Dexter Kozen* showed in [Koz00] that KAT subsumes propositional Hoare logic [Hoa69]. Additionally, it can be used to reason about programs [Koz03] [AHK07], and also when reasoning about concurrent actions is required, or in logic from complex contracts [PS12].

More recently KAT has been used to develop a formal system, namely *NetKAT*, to reason about packet switching networks [Koz14]. This system was presented by *Anderson, et al.* in [AFG$^+$14] and further developed by *Foster, et al.* in [FKM$^+$15] and has an important role in software-defined networks.

These are just some of the applications of KA. As one can see, it plays a major role in several different areas of computer science and therefore we encourage the reader to pursue more threads of investigation, so that one can have a wider view of all the implications of KA.

2.5 Related Work

There has been a lot of work done when it comes to decision procedures for KA. For instance, *Thomas Braibant* and *Damien Pous* implemented a new tactic in the `COQ` proof assistant to decide

Kleene Algebra [BP12]. Moreover, *Thierry Coquand* and *Vincent Siles* described and formally verified a procedure in [CS11] base on *Brzozowki*'s derivatives [Brz64]. Also *Alexander Krauss* and *Tobias Nipkow* came up with a decision procedure for regular expression equivalence and used it to prove equations in relation algebra in Isabelle/HOL [KN12]. *Marco Almeida, Nelma Moreira* and *Rogério Reis* developed an algorithm in [AMR10] that whithout constructing the underlying automata, decides the equivalence of regular expressions by testing the equivalence of their partial derivatives. Moreover, this alorithm was then implemented and verified in COQ by *Nelma Moreira, David Pereira* and *Simão Melo de Sousa* [MPMdS12]. Another example is the work of *Tobias Nipkow* and *Dmitriy Traytel* in [NT14] that formalises an unified framework for verified regular expression decision procedures.

When it comes to decision procedures based on bisimulations, we have the work of *Rat, J., et al.* in [RBR13] which uses bisimulation up to congruence, but differs from the one presented in [BP11] in the fact that they can deal with (quasi)-equations over arbitrary languages. Also, the work of *Marco Almeida, Nelma Moreira* and *Rogério Reis* in [AMR09] follows the work done by the same authors on [AMR08] where the proposed algorithm is closely related with the coalgebraic approach to the automata developed by *Rutten* in [Rut03].

2.6 Conclusions

This chapter introduced the reader to the notation used throughout this document. Additionally, it presented the reader with some fundamental definitions that will facilitate the understanding of the following chapters, and it also showed that Kleene Algebra plays a big role in several different areas of computer science.

In the next chapter we will discuss a decision procedure for SKA.

Chapter 3

A Decision Procedure for Synchronous Kleene Algebra

3.1 Introduction

This chapter is one of me most important ones in this document, since it guides the reader through the detail of the decision procedures for deciding Synchronous Kleene Algebra SKA. This decision procedure is a combination of two existing procedures, namely, an automaton construction from partial derivatives of a given regular expression [BCFM15], and the second one is an automata comparison using bisimulation up-to congruence [BP11].

This chapter comprises two more sections. Section 3.2 contains a detailed explanation of the decision procedure. The most important conclusions gathered from this chapter will be presented in section 3.4. The details regarding the implementation of this procedure will be discussed in chapter 4.

3.2 Decision Procedure

In a brief way, this procedure can be explained in the following way: given two regular expressions, we use the partial derivatives method to build two NFAs (one for each regular expression) that accept the languages generated by said regular expressions. Afterwards we use the bisimulation up-to congruence to compare both NFAs. In the next subsections we will cover the details of this procedure.

3.2.1 Regular Expression Processing

In order for one to build the NFA, it is needed to perform some operations on the regular expressions. In [BCFM15], *Broda, S., et al.* defined a formal product (see equation 3.1) and formal concatenation (see equation 3.2) between two SKA expressions. Moreover, they have some peculiarities, namely, products by "0" are not considered, and "1" remains as the identity function on both sides. Furthermore we refrained from writing $\alpha \otimes 1 = 1 \otimes \alpha = \alpha$ ($\alpha \otimes 1 = 1 \odot \alpha = \alpha$) in the cases that $\beta = 1$ on definition 3.

Definition 3. *Formal Product ($\otimes$) and Formal Concatenation ($\odot$)*

$$\alpha \otimes \beta = \begin{cases} \alpha \times \beta & \textit{if } \beta \neq 1 \\ \alpha & \textit{if } \beta = 1 \end{cases} \tag{3.1}$$

$$\alpha \odot \beta = \begin{cases} \alpha \cdot \beta & \textit{if } \beta \neq 1 \\ \alpha & \textit{if } \beta = 1 \end{cases} \tag{3.2}$$

As shown in [BCFM15], one can now define the same operations over sets of SKA expressions. Let A, B be sets such that $A, B \subseteq \mathcal{T}_{SKA}$, the concatenation and product operations are defined by $A \odot B = \{\alpha \odot \beta \mid \alpha \in A\backslash\{0\}, \beta \in B\backslash\{0\}\}$ and $A \otimes B = \{\alpha \otimes \beta \mid \alpha \in A\backslash\{0\}, \beta \in B\backslash\{0\}\}$, respectively.

The construction of the NFA is divided in two steps. For the first step, one needs to find the support set of a given regular expression. Furthermore, this support set is given by the function $\pi(\alpha)$ (see definition 4). The result set of this function becomes the set of states of the NFA.

Definition 4. *The function $\pi(\alpha)$ is inductively defined as follows:*

$$\begin{array}{ll} \pi(0) = \pi(1) = \emptyset & \pi(x) = \{1\}\ , x \in \mathcal{K}_b \\ \pi(\alpha^*) = \pi(\alpha) \odot \alpha^* & \pi(\alpha + \beta) = \pi(\alpha) \cup \pi(\beta) \\ \pi(\alpha\beta) = \pi(\alpha) \odot \beta \cup \pi(\beta) & \pi(\alpha \times \beta) = \pi(\alpha) \otimes \pi(\beta) \cup \pi(\alpha) \cup \pi(\beta) \end{array}$$

Before we can perform derivatives over a regular expression, we still need to define a function that checks for the existence of the empty string in the language generated by the given regular expression, returning $\top$ in the case of its presence, or $\bot$ otherwise. Since ε denotes the empty string in a language, the function name, $\varepsilon(\alpha)$, chosen by *Broda, S., et al.* in [BCM14], is quite formidable. For the formal definition of this function please refer to definition 5.

Definition 5. *Empty String Checking Function.*

$$\begin{array}{ll} \varepsilon(a) = \varepsilon(0) = \bot & \varepsilon(\alpha^*) = \varepsilon(1) = \top \\ \varepsilon(\alpha + \beta) = \varepsilon(\alpha) + \varepsilon(\beta) & \varepsilon(\alpha\beta) = \varepsilon(\alpha)\varepsilon(\beta) \\ \varepsilon(\alpha \times \beta) = \varepsilon(\alpha)\varepsilon(\beta) & \end{array}$$

The notion of partial derivatives of regular expressions was introduced by *Valentin Antimirov* in [Ant96], which elaborates on the work of *Brzozowski* in [Brz64]. At this point, the reader now possesses all the necessary knowledge to fully understand the partial derivative function. This function (see definition 6) returns the set of all partial derivatives of a SKA term α with respect to the letter $\sigma \in \Sigma$, denoted $\partial_\sigma(\alpha)$ by the original authors [BCM14],[BCFM15]. Please note that Σ is the powerset construction of all the symbols in $\mathcal{K}_b$ without the empty set, i.e. $\Sigma = \mathcal{P}(\mathcal{K})\backslash\{\emptyset\}$.

Definition 6. *Partial Derivative Function.*

$$\partial_\sigma(0) = \partial_\sigma(1) = \emptyset$$

$$\partial_\sigma(a) = \begin{cases} \{1\} & \textit{if } \sigma = \{a\} \\ \emptyset & otherwise \end{cases}$$

$$\partial_\sigma(\alpha^*) = \partial_\sigma(\alpha) \odot \alpha^*$$

$$\partial_\sigma(\alpha + \beta) = \partial_\sigma(\alpha) \cup \partial_\sigma(\beta)$$

$$\partial_\sigma(\alpha\beta) = \partial_\sigma(\alpha) \odot \beta \cup \varepsilon(\alpha) \odot \partial_\sigma(\beta)$$

$$\partial_\sigma(\alpha \times \beta) = (\textstyle\bigcup_{\sigma_1 \times \sigma_2 = \sigma} \partial_{\sigma_1}(\alpha) \otimes \partial_{\sigma_2}(\beta)) \cup \varepsilon(\alpha) \otimes \partial_\sigma(\beta) \cup \varepsilon(\beta) \otimes \partial_\sigma(\alpha)$$

The function presented on definition 6 will give us the set of transitions from a given state α with respect to the letter $\sigma \in \Sigma$.

3.2.2 Bisimulations and Coinduction

This subsection will give the reader with some background information on bisimulations and coinduction, in order to provide a better understanding of the *bisimulation up-to congruence* shown by *Filippo Bonchi* and *Damien Pous* on [BP11].

A bisimulation is nothing more than a way to define when two systems have the same behaviour with no regard for their internal structures. According to [BP11], a formal definition of bisimulation on states with a notion of progression is given by definition 7, where the notation $\delta_\sigma(x)$ represents the transition function, where given the letter $\sigma \in \Sigma$ and the state x as inputs, yields the next state. Moreover, $Final(x)$ is the function that outputs $\top$ or $\bot$ if the given state is either final or not final, respectively.

Definition 7. *Bisimulation. Given S a finite set of states and to relations R, R' over states such that $R, R' \subseteq S \times S$, R progresses to R' denoted $R \rightarrow R'$ if whenever $x \; R \; y$ then*

$$Final(x) = Final(y) \;\wedge\; \forall_\sigma \in \Sigma,\;\; \delta_\sigma(x) \; R' \; \delta_\sigma(y)$$

A bisimulation is a relation R such that $R \rightarrow R$.
Proposition 1. *Coinduction. Two states X and Y are language equivalent iff there exists a bisimulation that relates them, i.e. $X \sim Y$.*

With this concept in mind, we are now able to define a bisimulation up-to a given function on relations on S. For further insights into the following definitions we refer the reader to the original author's work [BP11] and [BP13].

Definition 8. *Bisimulation up-to f. Given a function $f:\ \mathcal{P}(S\times S)\rightarrow\mathcal{P}(S\times S)$ A relation R is a bisimulation up to f if $R\rightarrow f(R)$ if whenever $x\ R\ y$ then*

$$Final(x) = Final(y)\ \wedge\ \forall_\sigma\in\Sigma,\ \ \delta_\sigma(x)\ f(R)\ \delta_\sigma(y)$$

The reflexive (r), symmetric (s) and transitive (t) closure of a relation R, denoted $e(R)$ is inductively defined by rules 3.3 - 3.6, where id is the identity rule.

$$\frac{}{x\ e(R)\ x}r \quad (3.3)$$

$$\frac{x\ e(R)\ y \qquad y\ e(R)\ z}{x\ e(R)\ z}t \quad (3.5)$$

$$\frac{x\ e(R)\ y}{y\ e(R)\ x}s \quad (3.4)$$

$$\frac{x\ R\ y}{x\ e(R)\ y}id \quad (3.6)$$

In order to facilitate the understanding of *bisimulations up-to congruence* the reader must first interiorise the concept of *congruence closure* (see definition 9). We ask the reader to regard the operation $+$ as the set theoretic union of sets of states X and Y in $\mathcal{P}(S)$.

Definition 9. *Congruence Closure. Let $u:\mathcal{P}(\mathcal{P}(S)\times\mathcal{P}(S))\rightarrow\mathcal{P}(\mathcal{P}(S)\times\mathcal{P}(S))$ be a function on relations on sets of states defined as:*

$$u(R)\rightarrow\{(X_1+X_2,Y_1+Y_2)\mid X_1\ R\ Y_1\ \wedge\ X_2\ R\ Y_2\}$$

The function $c=(r\cup s\cup t\cup u\cup id)^\omega$ is called the congruence closure function.

The new rule u is defined as

$$\frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1+X_2\ c(R)\ Y_1+Y_2}u \quad (3.7)$$

As proven by *Filippo Bonchi* and *Damien Pous* on [BP11], any bisimulation up to c is contained in a bisimulation.

We will now present a variation of the *Hopcroft* and *Karp*'s algorithm [HK71] presented by *Filippo Bonchi* and *Damien Pous* on [BP11], for checking the equivalence of sets of states X and Y of a NFA on the fly.

On step $3b$ of the following algorithm, one has to check if some pair (X',Y') belongs to the congruence closure of the relation. *Bonchi* and *Pous* proposed that we look at each pair $(X,Y)\in R$ as a pair of rewriting rules such that $X\rightarrow X+Y$ and $Y\rightarrow X+Y$. These rules can be used to compute the normal forms for sets of states since by idempotence $X\ R\ Y$ entails $X\ c(R)\ X+Y$.

Algorithm 1 Up-to Congruence Equivalence Checking Algorithm.

```
1. $R, todo \leftarrow \emptyset$
2. insert pair $(X, Y)$ in $todo$
3. while $todo$ is not empty
   (a) extract $(X', Y')$ from $todo$
   (b) if $(X', Y') \in c(R \cup todo)$ then skip
   (c) else
           if $final(X') \neq final(Y')$ then return $false$
           else
               for all $\sigma \in \Sigma$
                   insert $(\delta_\sigma(X'),\ \delta_\sigma(Y'))$ in $todo$
               insert $(X', Y')$ in $R$
4. return true
```

Moreover, the normal form of a set is just the largest set of its equivalence class. Furthermore, *Bonchi* and *Pous* defined these concepts as shown in definitions 10 and 11.

Definition 10. *Let $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ be a relation on sets of states. The smallest irreflexive relation $\rightsquigarrow_R$, such that $\rightsquigarrow_R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ satisfies the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R\ X+Y} \quad (3.8) \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R\ X+Y} \quad (3.9) \qquad \frac{Z \rightsquigarrow_R Z'}{U+Z \rightsquigarrow_R U+Z'} \quad (3.10)$$

$X \downarrow_R$ denotes the normal form of a set X with respect to $\rightsquigarrow_R$

Definition 11. *For all relations R, and for all states $X, Y \in \mathcal{P}(S)$ we have $X \downarrow_R = Y \downarrow_R$ iff $(X, Y) \in c(R)$.*

With this information, step $3b$ becomes the computation of the normal forms of X' and Y' *w.r.t.* $R \cup todo$. Another way to look at it is by rewriting $X, Y \rightsquigarrow_R X + Y$ whenever $(X, Y) \in R$.

3.2.3 Decision Procedure

With the information provided in the previous subsections 3.2.1 and 3.2.2, the decision procedure is very simple to understand. One starts by constructing the automaton based on the partial derivatives (see definition 6) and the $\pi(\alpha)$ function (see definition 4) with respect to each regular expression given. Afterwards, one uses the algorithm 1 to check the equivalence of the resulting NFAs.

Due to lack of time, we did not perform a complexity analysis of this procedure, or provided a proof of correctness. However the complexity analysis for the automata comparison (*i.e.* algorithm 1) is provided in [BP13]. Furthermore we intend to conduct such analysis of the overall procedure as well as provide a proof of correctness in the near future.

3.3 Toy Example

For a better understanding of this decision procedure, we present the reader with a visual toy example. Let $\alpha = a^* \cdot b \cdot a^* \cdot b \cdot (a+b)^*$ and let $\beta = (a+b)^* \cdot b \cdot (a+b)^* \cdot b \cdot (a+b)^*$. From this point it is obvious to see that $\mathcal{K}_b = \{a, b\}$, and therefore $\Sigma = \{\{a\}, \{b\}, \{a, b\}\}$. We start by computing the support set of each expression, obtaining the results depicted on figure 3.1.

$$\pi(\alpha) = \{((((a^* \cdot b) \cdot a^*) \cdot b) \cdot (a+b)^*),\ ((a^* \cdot b) \cdot (a+b)^*),\ (a+b)^*\}$$

$$\pi(\beta) = \{(((((a+b)^* \cdot b) \cdot (a+b)^*) \cdot b) \cdot (a+b)^*),\ (((a+b)^* \cdot b) \cdot (a+b)^*),\ (a+b)^*\}$$

Figure 3.1: Result of functions $\pi(\alpha)$ and $\pi(\beta)$.

Now that one has the states for our automaton, one needs to find which transitions are valid, and for that we compute the partial derivatives. The partial derivatives of α and β with respect to every $\sigma \in \Sigma$ are shown in figure 3.2. Additionally, we compute the partial derivatives on the rest of the states (see figure 3.3). In order to facilitate the presentation, we say that $\gamma = ((a^* \cdot b) \cdot (a+b)^*)$, $\lambda = (((a+b)^* \cdot b) \cdot (a+b)^*)$ and $\mu = (a+b)^*$.

$$\begin{aligned}
&\partial_a(\alpha) = \{((((a^* \cdot b) \cdot a^*) \cdot b) \cdot (a+b)^*)\}\\
&\partial_b(\alpha) = \{((a^* \cdot b) \cdot (a+b)^*)\}\\
&\partial_{\{a,b\}}(\alpha) = \emptyset\\
&\partial_a(\beta) = \{(((((a+b)^* \cdot b) \cdot (a+b)^*) \cdot b) \cdot (a+b)^*)\}\\
&\partial_b(\beta) = \{(((((a+b)^* \cdot b) \cdot (a+b)^*) \cdot b) \cdot (a+b)^*),\ \ (((a+b)^* \cdot b) \cdot (a+b)^*)\}\\
&\partial_{\{a,b\}}(\beta) = \emptyset
\end{aligned}$$

Figure 3.2: Partial Derivitaves of α and β *w.r.t.* every $\sigma \in \Sigma$.

$$\begin{aligned}
&\partial_a(\gamma) = \{\gamma\}\\
&\partial_b(\gamma) = \{\mu\}\\
&\partial_{\{a,b\}}(\gamma) = \emptyset\\
&\partial_a(\lambda) = \{\lambda\}\\
&\partial_b(\lambda) = \{\lambda,\ \mu\}\\
&\partial_{\{a,b\}}(\lambda) = \emptyset\\
&\partial_a(\mu) = \{\mu\}\\
&\partial_b(\mu) = \{\mu\}\\
&\partial_{\{a,b\}}(\mu) = \emptyset
\end{aligned}$$

Figure 3.3: Partial Derivitaves of $\gamma,\ \lambda$ and μ *w.r.t.* every $\sigma \in \Sigma$.

After performing all the computations above, we obtain the resulting automaton depicted in figure 3.4.

At this point, we just have to execute the algorithm shown in 1. As depicted by figure 3.5, one starts by extracting (α, β) from $todo$, afterwards, since $Final(\alpha) = Final(\beta) = \bot$, one inserts $(\gamma, \beta + \lambda)$ in $todo$ and (α, β) in R. After step 3 the algorithm stops, yielding $true$.

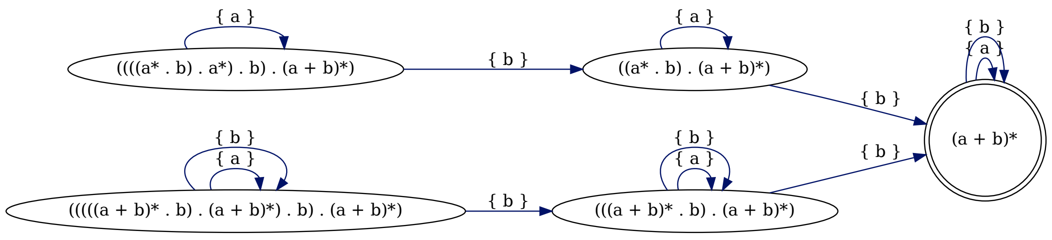
{ a }
((((a* . b) . a*) . b) . (a + b)*)
{ b }
{ a }
((a* . b) . (a + b)*)
{ b }
{ b }
{ a }
(a + b)*
{ b }
{ a }
(((((a + b)* . b) . (a + b)*) . b) . (a + b)*)
{ b }
{ b }
{ a }
(((a + b)* . b) . (a + b)*)
{ b }

Figure 3.4: Automaton Generated from α and β.

$\mathit{Step}\ 0$	$\rightarrow$	$\mathit{todo} = \emptyset$	$R = \emptyset$	$R \cup \mathit{todo} = \emptyset$
$\mathit{Step}\ 1$	$\rightarrow$	$\mathit{todo} = \{(\alpha, \beta)\}$	$R = \emptyset$	$R \cup \mathit{todo} = \{(\alpha, \beta)\}$
$\mathit{Step}\ 1.1$	$\rightarrow$	$\mathit{processing}\ \{(\alpha, \beta)\}$		
$\mathit{Step}\ 2$	$\rightarrow$	$\mathit{todo} = \{(\gamma, \beta + \lambda)\}$	$R = \{(\alpha, \beta)\}$	$R \cup \mathit{todo} = \{(\alpha, \beta), (\gamma, \beta + \lambda)\}$
$\mathit{Step}\ 2.1$	$\rightarrow$	$\mathit{processing}\ \{(\gamma, \beta + \lambda)\}$		
$\mathit{Step}\ 3$	$\rightarrow$	$\mathit{todo} = \{(\mu, \beta + \gamma + \mu)\}$	$R = \{(\alpha, \beta), (\gamma, \beta + \lambda\}$	$R \cup \mathit{todo} = \{(\alpha, \beta), (\gamma, \beta + \lambda), (\mu, \beta + \gamma + \mu)\}$

Figure 3.5: Computation of Algorithm 1 for α and β.

3.4 Conclusions

This chapter described a decision procedure for SKA based on the work of *Broda, S., et al.* in [BCFM15] and *Bonchi, F., et al.* in [BP11]. Furthermore we showed a toy example in order to facilitate the reader's comprehension of said procedure. In the next chapter we will discuss the implementation details of the aforementioned procedure using the `OCaml` programming language.

Chapter 4

Implementation and Validation

4.1 Introduction

In this chapter we will discuss the details regarding the implementation of the decision procedure explained in chapter 3. This chapter has the following organisation:

- Section 4.2 – **Technologies Used** – Explains in detail the most important technologies used throughout the implementation.
- Section 4.3 – **OCaml Implementation** – Presents the reader with the details of the OCaml implementation of this decision procedure.
- Section 4.4 – **Tests and Benchmarks** – Shows some performance and validation tests conducted under numerous conditions.
- Section 4.5 – **Conclusions** – Contains the most important conclusions drawn from this chapter.

4.2 Technologies Used

This section provides an insight into the technologies used throughout the implementations of the decision procedure explained in 3. The most important technologies used were OCaml which is described in subsection 4.2.1; OCamlgraph explained in subsection 4.2.2; and finally Menhir that can be seen in subsection 4.2.3.

4.2.1 OCaml

OCaml [XLR96] is a fully fledged, strongly typed functional programming language written in 1996 by Xavier Leroy, Jérôme Vouillon, Damien Doligez, and Didier Rémy, at INRIA in France [HMM14], [LDF^{+}18]. OCaml is also an imperative language, and also an object-oriented language. Moreover, it has a lot of useful libraries and features such as type inference and pattern matching for inductively defined types.

4.2.2 OCamlgraph

OCamlgraph is a generic graph library for the OCaml language [CFS07]. It provides the user with a set of graph data structures and generic implementations of common graph algorithms, written in way that is independent of the underlying graph data structure thanks to the use of functors.

4.2.3 Menhir

Menhir is a parser generator for the OCaml language [LDF+18]. It takes $LR(1)$ grammars as input and turns them into OCaml expressed parsers [PRG18b]. Furthermore, Menhir [PRG18a] was written by *François Pottier* and *Yann Régis-Gianas*, based on the $LR(1)$ parser construction technique by *Donald E.Knuth* [Knu65].

4.3 OCaml Implementation

In this section we will discuss how the decision procedure was implemented using the OCaml programming language. For a visual representation of the of the system, we ask the reader to look at figure 4.1. In order to process the desired regular expressions, we decided to implement a small parser which is discussed in subsection 4.3.1. Furthermore, constructing the automaton requires some operations over regular expressions, said functions are explained in subsection 4.3.2; subsection 4.3.3 contains the details of how we defined an automaton in OCaml. Moreover, the equivalence checker is explained in subsection 4.3.4. Please note that due to its extension, some of the `OCaml` code is located in appendix A.

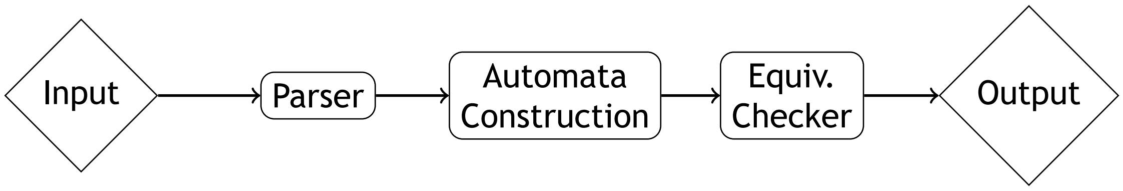

Figure 4.1: Visual Structure of the Implementation.

4.3.1 Regular Expression Parser

As most parsers, the one we implemented is divided in two stages, being the first one a lexical analyser (lexer) which asserts that we only read regular expressions containing the correct characters. The second one is a syntactical analyser (parser) which enforces us to read only expressions that are in accordance with the grammar rules. If any of these two fail, the user will be presented with an error message. This facilitates the fact that only *correctly* written regular expressions are given as input to the program.

We started this implementation by inductively defining what a regular expression can be i.e. its OCaml type, this definition can be seen on figure 4.2.

```
type regexp_t =
  | Epsilon
  | Empty
  | Char of char
  | Concat of regexp_t * regexp_t
  | Choice of regexp_t * regexp_t
  | Sync of regexp_t * regexp_t
  | Star of regexp_t
```

Figure 4.2: Regular Expression Type in OCaml.

The lexical rules were defined using all the regular operators seen in previous chapters of this document with the exception of the synchrony operator, which is now defined as ":". The complete definition can be seen in figure A.1.

The lexing stage splits the input into tokens, which are then transferred into the parsing stage. The latter is where one transforms these tokens into OCaml code. The syntactic rules for our regular expressions were defined as figure A.2 suggests. These rules are defined in a grammar style way, since this is the way *Menhir* accepts its rules.

For a better understanding of figure A.2, in line 1 we specify that our programs entry point will be "main" rule. This rule is then defined on line 6 which tells *Menhir* that our program consists of nothing more than a list of regular expressions followed by two characters, namely an End Of Line (EOL) character and an End Of File (EOF). Furthermore, on line 8 we define what a list of regular expressions is, and finally on line 13 we specify the syntactic rule for a well formed regular expression.

This parser is then invoked by the main function of our program which triggers the parsing process.

4.3.2 Operations over Regular Expressions

There are several operations over regular expressions defined by *Broda, S., et al.* in [BCFM15], and as they are an important part of the decision procedure, we had to implement such operations in OCaml. We shall start by defining the formal synchronous product and concatenation. The formal synchronous product (please refer to equation 3.1 on definition 3) is defined by the code shown in figure 4.3. In order to facilitate computations of such a product, we decided to implement an infix operator, namely $<:>$, such that $\alpha \otimes \beta \iff \alpha <:> \beta$.

The same formal synchronous product was defined for operations over sets, and its code is as shown in figure 4.4. This operation already makes use of the infix operator defined in figure 4.3, but for the set operations, we defined yet again other infix operator. To differentiate it from the previous one we added another ":" such that $<::>$ is new infix operator.

```
let (<:>) a b =
  match a,b with
  | Epsilon, Epsilon -> Set.singleton(Epsilon)
  | Epsilon, _ -> Set.singleton(b)
  | _, Epsilon -> Set.singleton(a)
  | _ -> Set.singleton(Sync(a,b))
```

Figure 4.3: Formal Synchronous Product in OCaml.

```
let (<::>) a b =
  Set.fold (fun a' acc -> match a' with
        Empty -> acc
      | _ -> Set.fold (fun b' acc2 -> match b' with
            Empty -> acc2
          | _ -> Set.union (a' <:> b') acc2) b acc) a Set.empty
```

Figure 4.4: Formal Synchronous Product for sets in OCaml.

The formal concatenation (see equation 3.2 on definition 3) was also defined both for single SKA terms as well as sets of terms. The operation for single SKA terms is shown in figure 4.5, which defines $< * >$ as the infix operator for this operation such that $\alpha \odot \beta \iff \alpha < * > \beta$. This same operation over sets defines $< ** >$ as its operator and can be seen in figure 4.6.

```
let (<*>) a b =
  match a,b with
    Epsilon, Epsilon -> Set.singleton(Epsilon)
  | Epsilon, _ -> Set.singleton(b)
  | _, Epsilon -> Set.singleton(a)
  | _ -> Set.singleton(Concat(a,b))
```

Figure 4.5: Formal Concatenation in OCaml.

```
let (<**>) a b =
  Set.fold (fun a' acc -> match a' with
        Empty -> acc
      | _ -> Set.fold (fun b' acc2 -> match b' with
            Empty -> acc2
          | _ -> Set.union (a' <*> b') acc2) b acc) a Set.empty
```

Figure 4.6: Formal Concatenation for sets in OCaml.

The implementation of the function that checks for the existence of the empty string in the language generated by a given regular expression (see definition 5) is given on figure A.3.

One of the most important functions in this implementation is the `derivate exp` x (see figure 4.7) which returns the set o all partial derivatives of a regular expression (`exp`) with respect to the letter "x". Furthermore, this function implements the partial derivatives defined in the previous chapter (see definition 6).

```
let rec derivate exp (x : char Set.t) =
  match exp with
    Char a when Set.singleton a = x -> Set.singleton (Epsilon)
  | Empty  | Epsilon | Char _ -> Set.empty
  | Choice (f,g) -> (derivate f x) ||. (derivate g x)
  | Concat (f,g) -> ((derivate f x) <**> Set.singleton(g))
                    ||. (Set.singleton(emptyWord f) <**> (derivate g x))
  | Star s ->  (derivate s x) <**> Set.singleton(Star s)
  | Sync(f,g) when Set.cardinal x = 1 -> ((derivate f x) <::> (derivate g x))
                    ||. (Set.singleton(emptyWord f) <::> (derivate g x))
                    ||. (Set.singleton(emptyWord g) <::> (derivate f x))
  | Sync(f,g) -> let ps = ((powerset x) |> Set.remove Set.empty |> Set.remove x) in
                    (Set.fold (fun c1 acc -> let c2 = x -. c1 in
                    ((derivate f c1) <::> (derivate g c2)) ||. acc) ps Set.empty )
                    ||. (Set.singleton(emptyWord f) <::> (derivate g x))
                    ||. (Set.singleton(emptyWord g) <::> (derivate f x))
```

Figure 4.7: Set of partial derivatives of a SKA term w.r.t. to a letter ”x” in OCaml.

4.3.3 Automata

The first step into automata implementation in OCaml was the creation of two modules. The first one would represent the Vertices, and the second one would represent the Edges (see figure 4.8).

```
(* representation of a vertex *)
module Node = struct
  type t = regexp_t
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end

(* representation of an edge *)
module Edge = struct
  type t = string
  let compare = Pervasives.compare
  let equal = (=)
  let default = ""
end
```

Figure 4.8: Modules for Vertex and Edge Representation.

In order to take advantage of some pre-defined functions over graphs, we decided to use OCaml-graph as a representation for our NFA. Adittionally we defined two more modules. Module `G` defines the type of graph we want to use, namely a bidirectional directed graph with labeled transitions (line 1 on figure A.4). Module `NFA` depicted by figure A.4, defines type `nft` where `size` represents the size of the automaton, `delta` specifies the transition function (which in fact is the NFA itself), and finally `accept` is the set of accepting states (i.e. final states).

Now that we have the NFA defined in OCaml, we have to build it according to a given regular expression. Moreover, we need to generate the set of states (vertices) and their transitions (edges). As explained in the previous chapter, the set of states is given by the $\pi(\alpha)$ (see definition

4) function which was implemented as shown in figure 4.9. For the edge generation, one first needs to determine which alphabet is used in the regular expression at hand. Furthermore, that alphabet is determined by the function `getSymbols exp` depicted on figure A.5. We are now able to fully understand the function that builds the automaton given a regular expression. This function is called `buildAutomata exp g` which accepts an expression and a graph. We refer the reader to figure 4.10 for its implementation.

```ocaml
(* Function that creates a support set for a given regular expression *)
let rec piFun exp =
  match exp with
    Empty | Epsilon -> Set.empty
  | Char _ -> Set.singleton(Epsilon)
  | Star s -> (piFun s) <**> (Set.singleton(Star s))
  | Choice(a,b) -> (piFun a) ||. (piFun b)
  | Concat(a,b) -> (piFun a) <**> Set.singleton(b) ||. (piFun b)
  | Sync(a,b) -> (piFun a) <::> (piFun b) ||. (piFun a) ||. (piFun b)
```

Figure 4.9: OCaml Implementation of $\pi(\alpha)$.

```ocaml
let rec getEdges exp src l g =
  if Set.is_empty exp then g
  else let e,s = pop exp in getEdges s src l (G.add_edge_e g (G.E.create src l e))

let buildAutomata exp g =
  (* let g = G.empty in *)
  let sigma = (powerset (getSymbols exp)) |> Set.remove Set.empty in
  let states = Set.singleton(exp) ||. piFun exp in
  Set.fold (fun x acc ->
      Set.fold (fun y acc2 ->
          Set.fold (fun z acc3 -> G.add_vertex acc3 (G.V.create z) ) (*Create Nodes *)
            states (getEdges (derivate y x) y (sprint_set ~first:"{" ~sep:", " ~last:"}"
             ↪  x) acc2) ) (*Create Edges*)
        states acc)
    sigma g
```

Figure 4.10: OCaml Function for Automaton Construction.

4.3.4 Equivalence Checker

The equivalence checking function is depicted in figure 4.11. In line 5 we check if both states x and y from automaton t are simultaneously final or not, *i.e.* if $Final(x) = Final(y)$. On line 6 we check if (x, y) belong to $c(R)$ and if so, we can skip it, otherwise we insert all pairs $(\delta_\sigma(x), \delta_\sigma(y))\ \forall_\sigma \in \Sigma$ in *todo*.

4.4 Tests and Benchmarks

As a way of validating the implementation, some tests were conducted. For all tests we start with two inputs A and B, and an expected output. Afterwards we register the given output and

```
let rec loop todo =
    match R.Q.pop todo with
    | None -> true
    | Some ((x,y),todo) ->
      if not (R.check t x y) then raise CE;
      if unify x y todo then loop todo
      else loop (push_span x y todo)
```

Figure 4.11: OCaml Function for Equivalence Checking.

match it against the expected output. Table 4.1 shows the results obtained.

Test Number	Input A	Input B	Output	Expected Output	Passed
1	$a+(b+c)$	$(a+b)+c$	$\top$	$\top$	✓
2	$a+b$	$b+a$	$\top$	$\top$	✓
3	$a+0$	a	$\top$	$\top$	✓
4	$a+a$	a	$\top$	$\top$	✓
5	$a.(b.c)$	$(a.b).c$	$\top$	$\top$	✓
6	$1.a$	a	$\top$	$\top$	✓
7	$a.1$	a	$\top$	$\top$	✓
8	$a.(b+c)$	$a.b+a.c$	$\top$	$\top$	✓
9	$(b+a)^*$	$(a+(b.b))^*$	$\perp$	$\perp$	✓
10	$(a.(b+a)^*)\times(a+(b.b))^*)$	$(c+a)^*$	$\perp$	$\perp$	✓
11	$a^*.b.a^*.b.(a+b)^*$	$(a+b)^*.b.(a+b)^*.b.(a+b)^*$	$\top$	$\top$	✓
12	$(a+b)^*.(a.a+b.b)$	$(a+b)^*.a.a+(a+b)^*.b.b$	$\top$	$\top$	✓
13	$a.b^*+a.c^*$	$a.(b^*+c^*)$	$\top$	$\top$	✓
14	$(a+b).c$	$(b+a).d$	$\perp$	$\perp$	✓
15	$(a+b)+c$	$(b+a)+d$	$\perp$	$\perp$	✓
16	$(a+b)+c$	$(b+a)+c$	$\top$	$\top$	✓
17	$(a+b).c$	$(b+a).c$	$\top$	$\top$	✓
18	$a\times(b\times c)$	$(a\times b)\times c$	$\top$	$\top$	✓
19	$a\times b$	$b\times a$	$\top$	$\top$	✓
20	$a\times 0$	$0\times a$	$\top$	$\top$	✓
21	$a\times 1$	$1\times a$	$\top$	$\top$	✓
22	$a\times a$	a	$\top$	$\top$	✓
23	$a\times(b+c)$	$a\times b+a\times c$	$\top$	$\top$	✓
24	$(a+b)\times c$	$a\times c+b\times c$	$\top$	$\top$	✓
25	$a^*\times b\times a^*\times b\times(a+b)^*$	$(a+b)^*\times b\times(a+b)^*\times b\times(a+b)^*$	$\perp$	$\perp$	✓

Table 4.1: List of Tests Performed and their results.

As a pure illustrative exercise we ran non SKA tests ($1-9, 13-19$) on a `Python` implementation by *Ferreira, M., et al.*, using the *FAdo* library [RM18] by *Reis, R.* and *Moreira, N.*. Moreover we did not run the tests involving the synchrony operator because we could not find the notation for its use. The time comparison is shown in table 4.2, where all times are in seconds.

	Real	User	System
OCaml	0.540	0.063	0.438
Python	0.960	0.422	0.500

Table 4.2: Ilustrative Time Comparison Between Implementations.

4.5 Conclusions

As seen in this chapter, the decision procedure described in chapter 3 was successfully implemented in `OCaml`. To summarise, we feed two inputs α and β to the parser, which transforms the given regular expressions into `OCaml` code. Afterwards we compute the support set and the partial derivatives for both expressions. The resulting automaton is given to the equivalence checker which outputs $\top$ when $\alpha \sim \beta$ or $\perp$ otherwise.

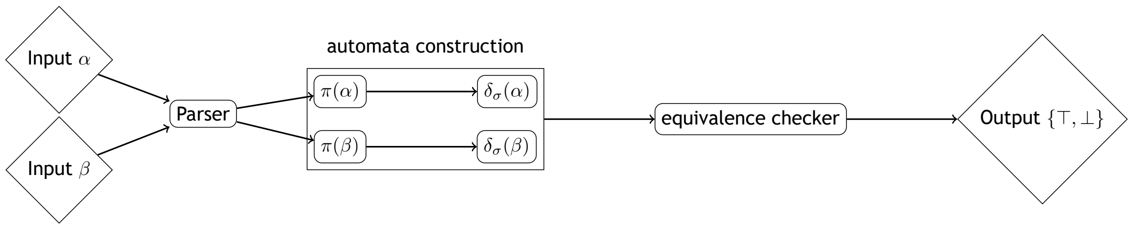

Figure 4.12: Visual Details of the Implementation.

Additionally some tests were performed in order to validate the implementation. Finally an illustrative comparison was made between a `Python` implementation and our `OCaml` implementation.

Chapter 5

Conclusions and Future Work

In this thesis it was presented a detailed explanation of a decision procedure for Synchronous Kleene Algebra SKA. This chapter contains the main conclusions drawn from the project, as well as it refers to some of the improvements and future work that the developer intends to complete in a near-future.

5.1 Main Conclusions

As it can be seen in this document, I successfully studied and implemented a decision procedure for Synchronous Kleene Algebra using `OCaml`. Additionally some tests were conducted in order to validate the implementation.

Some difficulties came up during the development of this thesis, however it is accurate to say that it was a very interesting and challenging assignment. For instance, due to the fact that there were a lot of new concepts to interiorise (*e.g.* bisimulation, coinduction), the time spent to investigate all of them was longer than expected. Familiarisation with `OCamlgraph` was also time consuming. Overcoming these issues was quite fulfilling and rewarding because it contributed to a deeper understanding of the concepts involved. This project not only introduced me to promising technologies such as `OCamlgraph` but also to the very fascinating field that is Synchronous Kleene Algebra.

To summarise, the main goals of this thesis were successfully achieved, and its development dazzled me into continuing to study and investigate further in this astonishing field that is Kleene Algebra.

5.2 Future Work

This thesis could be extended in the future in several different but complementary ways. When it comes to deciding SKA there is always room for improvement, and this thesis is no exception, and further decision procedures should be studied and implemented. Additionally, we would like to perform a complexity analysis on this procedure and provide a proof of correctness. Also, in the near-future it would be interesting to extend this project by extracting code generated from a constructive proof using the `COQ proof assistant` (see `https://coq.inria.fr/`).

Bibliography

[AFG+14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'14)*, pages 113-126, San Diego, California, USA, January 2014. ACM. 1, 9

[AHK07] Kamal Aboul-Hosn and Dexter Kozen. Local variable scoping and Kleene algebra with tests. *J. Log. Algebr. Program.*, 2007. 9

[AMR08] Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and mosses's rewrite system revisited. In Oscar H. Ibarra and Bala Ravikumar, editors, *Implementation and Applications of Automata*, pages 46-56, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 10

[AMR09] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing the equivalence of regular languages. In *Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems, DCFS 2009, Magdeburg, Germany, July 6-9, 2009.*, pages 47-57, 2009. Available from: `https://doi.org/10.4204/EPTCS.3.4`. 10

[AMR10] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing the equivalence of regular languages. *Journal of Automata, Languages and Combinatorics*, 15(1-2):7-25, 2010. Available from: `https://doi.org/10.25596/jalc-2010-007`. 10

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 - 319, 1996. Available from: `http://www.sciencedirect.com/science/article/pii/0304397595001824`. 12

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110-119, New York, NY, USA, 1979. ACM. Available from: `http://doi.acm.org/10.1145/567752.567763`. 9

[BCFM15] Sabine Broda, Sílvia Cavadas, Miguel Ferreira, and Nelma Moreira. Deciding synchronous kleene algebra with derivatives. In Frank Drewes, editor, *Implementation and Application of Automata*, pages 49-62, Cham, 2015. Springer International Publishing. ix, 1, 7, 8, 11, 12, 17, 21

[BCM14] Sabine Broda, Sílvia Cavadas, and Nelma Moreira. Derivative Based Methods for Deciding SKA and SKAT. Technical report, September 2014. 25p. Available from: `http://www.dcc.fc.up.pt/dcc/Pubs/TReports/TR14/dcc-2014-09.pdf`. 12

[BKM01] Roland Backhouse, Dexter Kozen, and Bernhard Möller, editors. *Applications of*

Kleene Algebra, number 01081 in Dagstuhl Seminar Reports, February 2001. 1

[BMR98] Thomas Brunn, Bernhard Möller, and Martin Russling. Layered graph traversals and hamiltonian path problems — an algebraic approach. In Johan Jeuring, editor, *Mathematics of Program Construction*, pages 96–121, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 1

[BP11] Filippo Bonchi and Damien Pous. Hopcroft and Karp's algorithm for Non-deterministic Finite Automata. Technical report, November 2011. 25p. Available from: `https://hal.archives-ouvertes.fr/hal-00639716`. ix, 1, 10, 11, 13, 14, 17

[BP12] Thomas Braibant and Damien Pous. Deciding kleene algebras in coq. *Logical Methods in Computer Science*, 8(1), 2012. Available from: `https://doi.org/10.2168/LMCS-8(1:16)2012`. 10

[BP13] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Principle of Programming Languages (POPL)*, pages 457–468, Roma, Italy, January 2013. ACM. 16p. Available from: `https://hal.archives-ouvertes.fr/hal-00639716v5`. 13, 15

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. Available from: `http://doi.acm.org/10.1145/321239.321249`. 10, 12

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In *Trends in Functional Programming (TFP'07)*, New York City, USA, April 2007. Available from: `http://www.lri.fr/~filliatr/ftp/publis/ocamlgraph-tfp-8.pdf`. 20

[Con12] J.H. Conway. *Regular Algebra and Finite Machines*. Dover Books on Mathematics. Dover Publications, 2012. Available from: `https://books.google.pt/books?id=KdtQAQAAQBAJ`. 6

[CS11] Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 119–134, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 10

[FKM$^+$15] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *Proc. 42nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'15)*, pages 343–355, Mumbai, India, January 2015. ACM. 1, 9

[HK71] John E. Hopcroft and R. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. 1971. Available from: `https://hdl.handle.net/1813/5958`. ix, 14

[HMM14] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. *Real World OCaml*. 2014. 19

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. Available from: `http://doi.acm.org/10.1145/363235.363259`. 9

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951. 7

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. 9

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956. vii, ix, xi, 6

[KN12] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, 49(1):95–106, Jun 2012. Available from: `https://doi.org/10.1007/s10817-011-9223-4`. 10

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965. Available from: `http://www.sciencedirect.com/science/article/pii/S0019995865904262`. 20

[Koz94] D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, May 1994. Available from: `http://dx.doi.org/10.1006/inco.1994.1037`. 6

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997. Available from: `http://doi.acm.org/10.1145/256167.256195`. 6, 7

[Koz00] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000. 9

[Koz03] Dexter Kozen. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Computer Science Department, Cornell University, November 2003. 6, 9

[Koz04] Dexter Kozen. Introduction to kleene algebra., 2004. [Online; `http://www.cs.cornell.edu/courses/cs786/2004sp/`, accessed December-2017]. 3, 5

[Koz14] Dexter Kozen. NetKAT: A formal system for the verification of networks. In Jacques Garrigue, editor, *Proc. 12th Asian Symposium on Programming Languages and Systems (APLAS 2014)*, volume 8858 of *Lecture Notes in Computer Science*, Singapore, November 17–19 2014. Asian Association for Foundation of Software (AAFS), Springer. 9

[LDF$^+$18] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and

Jérôme Vouillon. The OCaml system release 4.07: Documentation and user's manual. Intern report, Inria, July 2018. Available from: https://hal.inria.fr/hal-00930213. 19, 20

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983. Available from: http://www.sciencedirect.com/science/article/pii/0304397583901147. 8

[MPMdS12] Nelma Moreira, David Pereira, and Simão Melo de Sousa. Deciding regular expressions (in-)equivalence in coq. In Wolfram Kahl and Timothy G. Griffin, editors, *Relational and Algebraic Methods in Computer Science*, pages 98–113, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 10

[NT14] Tobias Nipkow and Dmitriy Traytel. Unified decision procedures for regular expression equivalence. In R. Gamboa and G. Klein, editors, *Interactive Theorem Proving (ITP 2014)*, volume 8558, pages 450–466, 2014. 10

[Pel85] D Peleg. Concurrent dynamic logic. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 232–239, New York, NY, USA, 1985. ACM. Available from: http://doi.acm.org/10.1145/22145.22172. 9

[Pra90] Vaughan Pratt. Dynamic algebras as a well-behaved fragment of relation algebras. In Clifford H. Bergman, Roger D. Maddux, and Don L. Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, pages 77–110, New York, NY, 1990. Springer New York. 9

[PRG18a] François Pottier and Yann Régis-Gianas. Menhir parser generator, 2018. [Online; http://gallium.inria.fr/~fpottier/menhir/, accessed 4-September-2018]. 20

[PRG18b] François Pottier and Yann Régis-Gianas. Menhir reference manual, 2018. [Online; http://gallium.inria.fr/~fpottier/menhir/manual.pdf, accessed 6-September-2018]. 20

[Pri10] Cristian Prisacariu. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming*, 79(7):608 – 635, 2010. The 20th Nordic Workshop on Programming Theory (NWPT 2008). Available from: http://www.sciencedirect.com/science/article/pii/S1567832610000391. vii, ix, xi, 1, 7, 8

[PS12] Cristian Prisacariu and Gerardo Schneider. A dynamic deontic logic for complex contracts. *The Journal of Logic and Algebraic Programming*, 81(4):458 – 490, 2012. Special Issue: NWPT 2009. Available from: http://www.sciencedirect.com/science/article/pii/S1567832612000197. 9

[RBR13] Jurriaan Rot, Marcello Bonsangue, and Jan Rutten. Coinductive proof techniques for language equivalence. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 480–492, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 10

[RM18] Rogério Reis and Nelma Moreira. Fado: tools for formal languages manipulation., 2018. [Online; `http://fado.dcc.fc.up.pt/`, accessed November-2017]. 25

[Rut03] J.J.M.M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1):1 – 53, 2003. 10

[XLR96] Damien Doligez Xavier Leroy, Jérôme Vouillon and Didier Rémy. Ocaml, 1996. [Online; `http:/ocaml.org` accessed 23-March-2018]. 19

Appendix A

OCaml Source Code

A.1 OCaml Source Code

```
let space = [' ' '\t']
rule token = parse
  | space          { token lexbuf }
  | '\n'           { new_line lexbuf; EOL }
  | ['a'-'z'] as c { CHAR c }
  | '0'            { EMPTY }
  | '1'            { EPSILON }
  | '*'            { STAR }    (* the Kleene star operator *)
  | '+'            { CHOICE }  (* the union operator *)
  | '.'            { CONCAT }  (* the concatenation operator *)
  | ':'            { SYNC }    (* the synchronous operator *)
  | '('            { LPAR }
  | ')'            { RPAR }
  | eof            { EOF }
  | _ as st        { lex_error lexbuf (String.make 1 st)}
```

Figure A.1: Lexer rules for Regular Expressions in OCaml.

```
%start main
%type <RegExp.regexp_t list> main

%%

main: r = regexlist EOL* EOF { List.rev r }

regexlist:
  r = regex {[r]}
| rl = regexlist  EOL+ r = regex  {r::rl}
| rl = regexlist EOL+ error  { print_syntax_error ($startpos($3)) ; rl }

regex:
  | EPSILON   { Epsilon }
  | EMPTY     { Empty }
  | c = CHAR  { Char c }
  | LPAR r = regex RPAR { r }
  | a = regex CHOICE b = regex { Choice(a, b) }
  | a = regex CONCAT b = regex { Concat(a, b) }
  | r = regex STAR { Star r }
  | a = regex SYNC b = regex { Sync(a,b)}
```

Figure A.2: Parser rules for Regular Expressions in OCaml.

```
let rec emptyWord w =
  match w with
    Empty | Char _ -> Empty
  | Epsilon | Star _ -> Epsilon
  | Choice (f,g) -> (match (emptyWord f) with
        Empty -> emptyWord g
        | Epsilon -> Epsilon
        | _ -> assert false)
  | Concat (f,g) | Sync(f,g) -> (match (emptyWord f) with
        Empty -> Empty
        | Epsilon -> emptyWord g
        | _ -> assert false)
```

Figure A.3: Nullable SKA term in OCaml.

```
module G = Graph.Persistent.Digraph.ConcreteBidirectionalLabeled(Node)(Edge)
module NFA = struct
  include G
  type nft = {
    size: int;
    delta: G.t;
    accept: regexp_t Set.t;  }
  let size a = G.nb_vertex a.delta
  let accept a = fold_vertex (fun x acc ->
        if emptyWord x = Epsilon then Set.add x acc else acc) a.delta Set.empty
  let vars a = let r =
        fold_edges_e (fun x acc -> Set.add (G.E.label x) acc) a.delta Set.empty in r
  let delta a v x =
        try
            fold_succ_e (fun y acc -> if G.E.label y = v then Set.add (G.E.src y) acc
                else acc ) a.delta x Set.empty
        with Invalid_argument _ -> Set.empty
  let delta_set a v x = Set.fold (Set.union % delta a v) x Set.empty
end
type nfa = NFA.nft
```

Figure A.4: OCaml NFA Implementation Module.

```
(* Function that returns a set of symbols in a given regular expression *)
let rec getSymbols exp =
  match exp with
    Empty | Epsilon-> Set.empty
  | Char c -> Set.singleton(c)
  | Choice (f,g) | Concat (f,g) | Sync (f,g) -> (getSymbols f) ||. (getSymbols g)
  | Star s -> getSymbols s
```

Figure A.5: Function that Exctracts the Symbols used in a Regular Expression.