# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sochor** Jméno: **Miroslav** Osobní číslo: **457870**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra teoretické informatiky**

Studijní program: **Informatika**

Studijní obor: **Teoretická informatika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Implementace větvících regulárních výrazů**

Název bakalářské práce anglicky:

**Implementation of forkable regular expressions**

Pokyny pro vypracování:

1) Research forkable regular expressions.2) Research the method for transforming regular expressions into the finite automata using Brzozowski derivatives.3) Design and implement the method for transforming forkable regular expressions into finite automata in Java programming language.4) Analyze the possibility of extending both the forkable regular expressions and the method for their transformation into automata.5) Test the implementation using appropriate forkable regular expressions.

Seznam doporučené literatury:

Will be provided by the supervisor.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Radomír Polách, katedra teoretické informatiky FIT**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **22.01.2018** Termín odevzdání bakalářské práce: ______________

Platnost zadání bakalářské práce: ______________

| | | |
|---|---|---|
| Ing. Radomír Polách<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | doc. RNDr. Ing. Marcel Jiřina, Ph.D.<br>podpis děkana(ky) |

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

| | |
|---|---|
| Datum převzetí zadání | Podpis studenta |

© ČVUT v Praze, Design: ČVUT v Praze, VIC

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

# Implementation of forkable regular expressions

***Miroslav Sochor***

Department of Theoretical Informatics
Supervisor: Ing. Radomír Polách

May 13, 2018

# Acknowledgements

I would like to thank my supervisor Ing. Polách for the chance to work on this topic. I would also like to thank my friends and family for their support.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 13, 2018 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Miroslav Sochor. All rights reserved.
*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

## Citation of this thesis

Sochor, Miroslav. *Implementation of forkable regular expressions.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

# Abstrakt

Větvící regulární výrazy jsou regulární výrazy rozšířené o unární operátor Fork. Tento operátor přidává souběžnou složku k výrazům. Cílem práce je tyto Větvící regulární výrazy implementovat, aby se daly využít k analýze paralelních programů. V práci jsou Větvící regulární výrazy rozšířeny o operátory Atomic, Sync a Async a převedeny metodou Brzozowského derivátů na konečný automat. Tento automat reprezentuje chování programu popsaného výrazem.

**Klíčová slova** větvící výrazy, deriváty, souběžnost, regulární výrazy, automaty

# Abstract

Forkable regular expressions are regular expressions extended with unary operator Fork. Fork adds concurrent part to the expressions. The goal of this thesis is to implement Forkable regular expressions, so that they can be used to analyze parallel programs. This thesis extends Forkable regular expressions with operators Atomic, Sync and Async. These extended Forkable regular expressions are transformed into a finite-state machine using Brzozowski derivatives. This automaton represents effects of a program described by the expression and its behavior.

**Keywords** forkable expressions, derivatives, concurrency, regular expressions, automata

# Contents

# List of Figures

# Introduction

The world is filled with events and activities happening concurrently. But in fact their effects are always sequenced by some means. Concurrent constructs are introduced into programs and with them some problems may arise. Problems such as locks and race conditions, that create non-deterministic behavior, must be recognized and resolved. Analyzing concurrency is hard but very useful.

On the other hand regular expressions are easy to understand and use. They are capable of modeling concurrency, but they do not do it very well. For this reason many extensions of regular expressions came into existence. One of these extended definitions is Forkable Regular Expressions (FREs) by Sulzmann and Thiemann that added operator *Fork*. *Fork* describes effects of creating a new thread and thus appends concurrent part to the expression.

The problem is that this added *Fork* operator may allow creation of non-regular languages. This can be avoided by restricting the use of *Fork* operator. Furthermore FREs lack synchronization events and thus cannot describe many parallel programs, or they do so very crudely. By adding other operators FREs can come closer to real parallel programs while maintaining their clarity.

The goal of this thesis is to add other operators that can bring FREs closer to parallel programs then implement a method to transform FREs according to the Sulzmann and Thiemann definition extended by these new operators into a finite state machine using Brzozowski derivatives. The programming will be done in the Java Programming Language.

Chapter 1

# Preliminaries

Basic definitions were adapted from [1] with some minor changes.

**Definition 1.** An alphabet, often denoted as $\Sigma$, is a set of symbols.

String $s \in \Sigma^*$ is a sequence of symbols of any finite length.

A string of zero length is called empty string and is denoted by $\epsilon$.

Language $L \subseteq \Sigma^*$ is a set of strings.

In the context of this thesis a symbol generally represents a primitive event such as reading or writing to memory. A string then describes a task.

## 1.1 Finite automaton

**Definition 2.** [1, 2] Finite automaton $A$ is a five-tuple, $A = (Q, \Sigma, \Delta, q_0, F)$, where:

$Q$ is a finite set of states,

$\Sigma$ is a finite set of input symbols i. e. an alphabet,

$\Delta$ is a transition relation, $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$, that links combination of a state and a symbol to a subset of next states,

$q_0$ is a start state, one of the states in $Q$,

$F$ is a set of final states or accepting states, subset of $Q$.

**Definition 3.** Let $A = (Q, \Sigma, \Delta, q_0, F)$ be a finite automata.
Configuration of $A$ is pair $(q, w) \in Q \times \Sigma^*$. Configuration $(q_0, w)$ is called starting configuration of $A$ and configuration $(q', \epsilon)$, where $q' \in F$, is called final or accepting configuration of $A$.

**Definition 4.** Let $A = (Q, \Sigma, \Delta, q_0, F)$ be a finite automata and let $\vdash_A \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ be a binary relation on a set of configurations of $A$, such that

$$(q, w) \vdash_A (q', a \cdot w) \iff (q, a, q') \in \Delta\,, \quad a \in \Sigma \cup \{\epsilon\}\,. \tag{1.1}$$

A member of $\vdash_A$ is called a transition in $A$.

**Definition 5.** Finite automaton $A$ is called deterministic, iff for every pair $(q, x) \in Q \times \Sigma$ exists at most one transition $((q, x \cdot w), (q', w))$ in $A$ for any $w \in \Sigma^*$. In other words, relation $\Delta$ can be substituted with function $\delta$, $\delta : Q \times \Sigma \to Q$.
If finite automaton is not deterministic, it is called non-deterministic.

Transition relation specifies triplets of start state, symbol or $\epsilon$, end state. Restriction to function means that every combination of a start state and a symbol gives just one end state.

**Definition 6.** Finite automaton $A = (Q, \Sigma, \Delta, q_0, F)$ accepts string $w \in \Sigma^*$ iff $(q_0, w) \vdash_A^* (q, \epsilon)$, where $q \in F$. In other words, there exists a sequence of transitions from the starting configuration to the accepting configuration.

Finite automaton $A$ accepts language $L$ iff for every string $w \in L$, $A$ accepts $w$.

**Definition 7.** Language $L$ is called regular language iff there exists a finite automaton $A$, such that $A$ accepts $L$.

Definitions are slightly modified from [1] and [2].

Regular languages are closed under the operations of union, concatenation and intersection as shown in [2].

## 1.2 Regular expressions

**Definition 8.** [1] Regular expressions $R$ denoting language $L(R)$ over $\Sigma$ are defined recursively as follows:

(i) $\emptyset$, $\epsilon$ and every symbol $a \in \Sigma$ are regular expressions denoting languages as follows:

$$\begin{aligned} L(\emptyset) &= \emptyset\,, \\ L(\epsilon) &= \epsilon\,, \\ L(a) &= a\,. \end{aligned} \tag{1.2}$$

(ii) If $S$ and $T$ are regular expressions then with ascending priority:

a) alternation $R = S + T$ is a regular expression denoting union of languages:

$$L(R) = L(S) \cup L(T)\,, \tag{1.3}$$

b) concatenation $R = S \cdot T$ is a regular expression denoting concatenation of languages (the dot between expressions can be omitted, if the meaning stays clear):

$$L(R) = L(S) \cdot L(T)\,, \tag{1.4}$$

c) iteration or Kleene star operation $R = S*$ is a regular expression denoting closure of language:

$$L(R) = \epsilon + L(S) \cdot L(S*) \tag{1.5}$$

d) parenthesis $R = (S)$ is a regular expression denoting the same language.

**Definition 9.** [1] Regular expressions $R$ and $S$ are equivalent iff they describe the same language:

$$R = S \iff L(R) = L(S)\,. \tag{1.6}$$

Derivatives of regular expressions were introduced by Brzozowski in [3]. Brzozowski uses regular expression and regular language very interchangeably, therefore a reformulation to fit regular expression definition was required.

**Definition 10.** [3] Let $R$ and $S$ be regular expressions and $x \in \Sigma^*$ a string of finite length.
If $L(S) = \{w \mid x \cdot w \in L(R)\}$, then $S$ is derivative of $R$ with respect to $x$ and is denoted as $d_x(R)$.

The process of finding a derivative is called derivation. It is easy to see that for every $R$ and $x$ all the possible derivatives of $R$ with respect to $x$ are equivalent.

**Definition 11.** [4] Left quotient of $L_2$ with $L_1$ is denoted as $L_1 \setminus L_2$, where

$$L_1 \setminus L_2 = \{w \mid \exists v \in L_1, v \cdot w \in L_2\}\,. \tag{1.7}$$

$x \setminus L$ is shortened notation of $\{x\} \setminus L$.

It is easy to see, that a derivative $S$ of $R$ with respect to $x$ can be interpreted as

$$L(S) = x \setminus L(R)\,. \tag{1.8}$$

Brzozowski devised a recursive method to find derivatives of expressions:

$$\begin{aligned}
d_x(x) &= \epsilon\,,\\
d_x(y) &= \emptyset \quad \text{for } y \in \Sigma \cup \{\emptyset, \epsilon\} \text{ and } y \neq x\,,\\
d_x(R*) &= d_x(R) \cdot R*\,,\\
d_x(R \cdot S) &= d_x(R) + \delta(R) \cdot d_x(S)\,,\\
d_x(R + S) &= d_x(R) + d_x(S)\,,\\
d_\epsilon(R) &= R\,.
\end{aligned} \tag{1.9}$$

**Definition 12.** [3] Let R be a regular expression, then null function is defined as follows:

$$\delta(R) = \begin{cases} \epsilon & \text{if } \epsilon \in L(R) \\ \emptyset & \text{otherwise} \end{cases} \quad . \tag{1.10}$$

Null function is useful for finding a derivative of expression as shown in 1.9.

# Chapter 2

# State-of-the-art

At the beginning of this chapter in section 2.1, two basic ways of transforming a regular expression into a finite automaton are presented. A short summary of thoughts and algorithms about concurrency, that are the most relevant to this thesis, follow in section 2.2. In section 2.3 there is a terse analysis of created extensions of regular expressions.

## 2.1 Transformation of regular expressions into finite automata

It is easy to see that deterministic automata are easier to work with, and therefore a deterministic finite automaton is set as a transformation goal.

It has been proved in [5] that transforming regular expressions or non-deterministic finite automata into minimal deterministic finite automata is *PSPACE* problem.

### 2.1.1 Construction by structural induction

First to mention a straight-forward construction by induction. The construction of the automaton begins with one starting and one final state. Depending on the expression $R$, one of the following from figure 2.1 is chosen:

(a) if $R = \emptyset$

(b) if $R = \epsilon$

(c) if $R = x$

(d) if $R = S + T$

(e) if $R = S \cdot T$

(f) if $R = S*$

This approach generates a non-deterministic finite automaton with $\epsilon$ transitions, which has to be determinized afterwards.

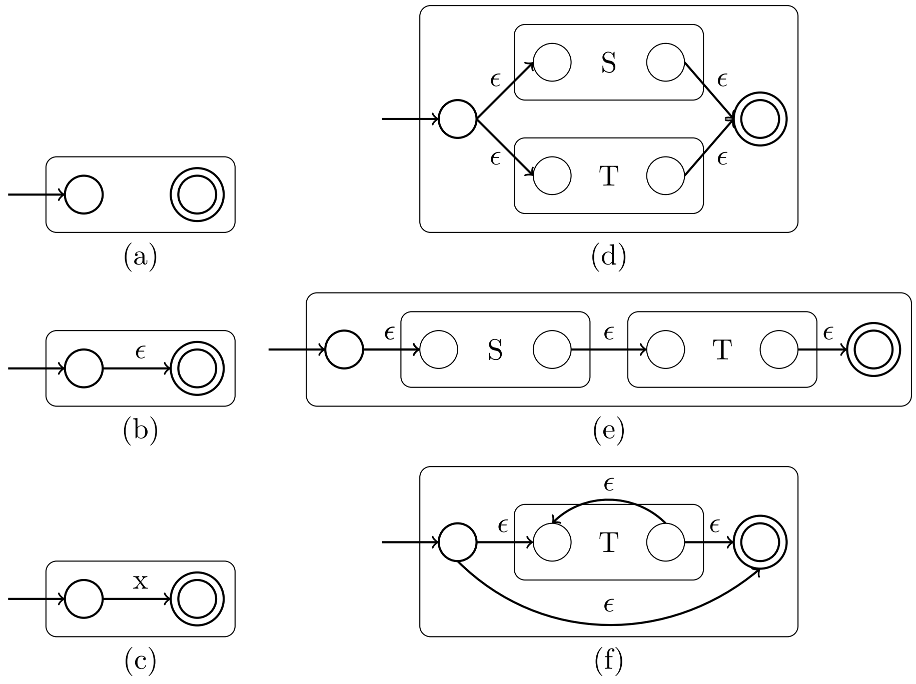

Figure 2.1: Induction steps in creating finite automaton [1]

### 2.1.2 Construction using Brzozowski derivatives

Other possible approach is to use Brzozowski derivatives.

A regular expression $R$ can be considered to be a state. A derivative of $R$ with respect to a string $x$ can be seen as a state $d_x(R)$ reached from state $R$ by a sequence of transitions using $x$. It means that the set of states $Q$ would contain $R$ and all its distinct (not equal) derivatives. Moreover every expression $R$, such that $\epsilon \in L(R)$, can be considered to be an accepting state.

As there is a finite number of not equal derivatives, this approach is possible. Brzozowski proved that this approach creates a minimal deterministic finite automaton and also that "*it is often quite difficult to determine whether two regular expressions are equal*". Therefore similarity, that can be decided in polynomial time, has been introduced in place of equality. Two similar expressions must be equal, even though there can be two dissimilar but equal expressions. Brzozowski proved that even the number of dissimilar derivatives is finite. [3]

An implementation of this method would iterate over existing derivatives trying to find a new one using (1.9). Accepting state can be checked using null function as in (1.10).

## 2.2 Concurrency analysis

Program analysis can be divided into two categories: *static analysis* and *dynamic analysis.* [6]

"*Static concurrency analysis is a technique for determining all the possible synchronization patterns in a concurrent program, without program execution. It is capable of detecting, for instance, infinite waits and concurrent updates to shared variables.*" However this static concurrency analysis is an *NP-hard* problem. [7] Detecting locks and other static analysis decision problems are *NP-complete.* [8, 9]

### 2.2.1 Taylor's algorithm

One of the algorithms for static concurrency analysis is described by Taylor in [9] and can be summarized as follows.

It needs a *call graph*, *scope information* and a *control flow graph* of the analyzed program. The algorithm generates a complete *concurrency history* of a program while reporting all synchronization events, all parallel and all infinite wait situations or *locks.* However as a static analysis tool it considers all paths executable and therefore it can report some events as suspicious even though they are impossible to occur. A *concurrency history* is a sequence of *concurrency states.* A *concurrency state* represents a status of *tasks* in a flow graph and a set of its successors is defined. More detailed information about analysis is disclosed in [9].

Young and Taylor complemented this algorithm with *symbolic execution* in [7]. Symbolic execution prunes out impossible situations and with them falsely reported events, while concurrency analysis reduces the number of *interleavings* for symbolic execution to explore. Pruning all impossible situations is similar to the *halting problem*, however it can still point to many unexecutable paths. [7] There are more optimizations in [7] to battle the *combinatorial explosion* that is inherent in the static concurrency analysis and symbolic execution itself.

### 2.2.2 Duesterwald and Soffa algorithm

The algorithm described by Duesterwald and Soffa in [8] uses *data-flow analysis* instead of *state based analysis* as Taylor in [9]. Data-flow based analysis has lower computational complexity but pays for it with less precision.

The algorithm also considers procedures and with them – possibly infinite – nesting of parallelism. To accommodate this they introduced *Module Interaction Graph* that splits the program into regions of statements. The ordering of regions is created by iterating on this graph using data-flow and thus regions capable of executing concurrently can be located.

### 2.2.3 Sinha and Wang algorithm

Sinha and Wang in [10] criticize the *redundant bi-modal reasoning* of other analyzers. By bi-modal reasoning they mean changing semantics from sequential (*intra-thread*) to concurrent (*inter-thread*) and back, which is the source of inefficiency. They proposed a two-stage program that separates the *intra-thread* and the *inter-thread*. The first stage summarizes the sequential part of every task and the second stage deals with concurrency. They also defined *concurrent control flow graphs* for concurrent programs as an extension of control flow graphs to bolster their program.

### 2.2.4 Amtoft's behaviors

In [11], Amtoft et al. defined a process algebra called *behaviors*, that can be constructed directly from CML (standard ML with added concurrency). These behaviors represent a communication topology between created concurrent tasks. Analysis is then done on behaviors. For a given behavior their algorithm produces some constraints that have to be solved. The solution then grants insight on the concurrency of analyzed behaviors.

## 2.3 Extensions of regular expressions

### 2.3.1 Constrained Expressions

Constrained expressions were created to provide an automated support for the design of concurrent systems. They can be divided into *system expressions* and *constraints.* [12]

*System expressions* are basically regular expressions with newly introduced operator *shuffle.*

**Definition 13.** [12, 13] Let $x, y \in \Sigma$ be symbols, $v, w \in \Sigma^*$ strings, $U, V \subseteq \Sigma^*$ sets of strings and $R, S, T$ regular expressions.
Operator shuffle $\parallel$ describes interleaving of two strings:

$$\begin{gathered} \epsilon \parallel x = x \parallel \epsilon = \{x\}\,, \\ x \cdot v \parallel y \cdot w = x \cdot (v \parallel y \cdot w) \cup y \cdot (x \cdot v \parallel w)\,. \end{gathered} \tag{2.1}$$

It can be lifted from strings to sets of strings:

$$U \parallel V = \{w \mid u \in U, v \in V, w \in u \parallel v\}\,, \tag{2.2}$$

then it can be used as regular expression operator:

$$R \parallel S = T \iff L(R) \parallel L(S) = L(T)\,. \tag{2.3}$$

This corresponds to sequencing two parallel tasks as an interleaving happens naturally and randomly. Also it is worth noticing that this construction does not increase the expressiveness of regular expressions as it can be substituted by – albeit long – sequence of alternations and concatenations [13].

*Constraints* do not need *shuffle operator*, instead they use operator *iterated shuffle.*

**Definition 14.** [12] Let $R$ be a regular expression.
Iterated shuffle (or shuffle closure) † is defined recursively as follows:

$$R\dagger = \epsilon + R \parallel R\dagger \ . \tag{2.4}$$

It is worth noting that the use of the *iterated shuffle* can describe a non-regular language as can be seen from expression $(a \cdot b \cdot c)\dagger$.

### 2.3.2 Concurrent regular expressions

Garg and Ragunath in [13] define Concurrent Regular Expression (CRE) by adding operators:

- *interleaving*,
- *$\alpha$-closure*,
- *synchronous composition*
- *renaming.*

Interleaving is equivalent to the *shuffle operator* 13 and $\alpha$-closure is equivalent to *iterated shuffle* 14.

**Lemma 1.** *[13] Shuffle satisfies the following properties:*

$$\begin{aligned} A \parallel B &= B \parallel A \quad \textit{(Commutativity)}\,,\\ A \parallel (B \parallel C) &= (A \parallel B) \parallel C \quad \textit{(Associativity)}\,,\\ A \parallel \{\epsilon\} &= A \quad \textit{(Identity of } \parallel\textit{)}\,,\\ A \parallel \emptyset &= \emptyset \quad \textit{(Zero of } \parallel\textit{)}\,,\\ (A + B) \parallel C &= (A \parallel C) + (B \parallel C) \quad \textit{(Distributivity over } +\textit{)}\,. \end{aligned} \tag{2.5}$$

Synchronous composition can be seen on figure 2.2. Because the use of the *iterated shuffle* can generate a non-regular language, Garg and Ragunath defined *concurrent regular languages* that CREs characterize. *Concurrent regular languages* are equivalent with *petri net languages*, because CRE can be transformed into a petri net and the other way around as Garg and Ragunath show in [13].

$$A \amalg B = \{w, w/\Sigma_A \in A, w/\Sigma_B \in B\} \quad \text{where } A \text{ and } B \text{ are sets}$$
$$\text{and } w/S \text{ denotes the restriction of string } w \text{ to the symbols in set } S$$

Figure 2.2: Definition of synchronous composition [13]

Chapter 3

# Forkable regular expressions

## 3.1 Original definition

The syntax definition of the Forkable Regular Expression (FRE) via regular grammar

$$R ::= \emptyset \mid \epsilon \mid x \mid R + R \mid R \cdot R \mid R* \mid Fork(R) \mid (R) \tag{3.1}$$

reveals a new operator. From definition 8 it adds the operator *Fork*, that took a function pattern.

**Definition 15.** [4] Trace language $L(R) \subseteq \Sigma^*$ and $L(R, K) \subseteq \Sigma^*$ is defined as

$$\begin{aligned}
L(R) &= L(R, \{\epsilon\})\,, & (3.2)\\
L(\emptyset, K) &= \emptyset\,, & (3.3)\\
L(\epsilon, K) &= K\,, & (3.4)\\
L(x, K) &= \{x\} \cdot K\,, & (3.5)\\
L(R + S, K) &= L(R, K) \cup L(S, K)\,, & (3.6)\\
L(R \cdot S, K) &= L(R, L(S, K))\,, & (3.7)\\
L(R*, K) &= \mu(\lambda X.L(R, X) \cup K)\,, & (3.8)\\
L(Fork(R), K) &= L(R, \{\epsilon\}) \parallel K)\,, & (3.9)
\end{aligned}$$

with respect to continuation language $K \subseteq \Sigma^*$.

The continuation language $K$ essentially contains everything that happens after the expression.

The rule for Kleene star (3.8) uses the least fixed point of the lambda function $\lambda X.L(R, X) \cup K)$. The least fixed point of that lambda function exists as shown in [4]. For full definition of the least fixed point operator $\mu$

see [14]. For purposes of this thesis, it is equal to the transitive closure and the rule (3.8) has the same meaning as recursive $L(R*, K) = L(R, L(R*, K)) \cup K$.

In the rule for *Fork* (3.9), there is once again the *shuffle operator* $\|$ from definition 13. The rule describes the behavior of the *Fork* operator. *Fork* causes interleaving of its operand with the continuation language.

**Definition 16** (Semantic equality)**.** [4] FREs $R$ and $S$ are equal, if their trace languages are equal with respect to any continuation language $K$:

$$R \equiv S \longleftarrow L(R, K) = L(S, K), \quad \forall K \in \Sigma^* . \tag{3.10}$$

They also introduced the *sequential part* $S(R)$ and the *concurrent part* $C(R)$ of FRE $R$ described by figure 3.1. *Sequential part* describes what happens in this task and *concurrent part* describes newly created concurrent tasks. [4] This corresponds to *intra-thread* and *inter-thread* from Sinha and Wang in [10].

$$\begin{aligned}
C(\emptyset) &= \emptyset\,, & S(\emptyset) &= \emptyset\,,\\
C(\epsilon) &= \epsilon\,, & S(\epsilon) &= \emptyset\,,\\
C(x) &= \emptyset \quad x \in \Sigma\,, & S(x) &= x \quad x \in \Sigma\,,\\
C(R+S) &= C(R) + C(S)\,, & S(R+S) &= S(R) + S(S)\,,\\
C(R \cdot S) &= C(R) \cdot C(S)\,, & S(R \cdot S) &= S(R) \cdot S + C(R) \cdot S(S)\,,\\
C(R*) &= C(R)*\,, & S(R*) &= C(R) * \cdot S(R) \cdot R*\,,\\
C(Fork(R)) &= Fork(R)\,, & S(Fork(R)) &= \emptyset\,,
\end{aligned}$$

Figure 3.1: Concurrent and sequential recursive function [4]

$$d_x(\emptyset) = \emptyset\,, \tag{3.11}$$

$$d_x(\epsilon) = \emptyset\,, \tag{3.12}$$

$$d_x(y) = \begin{cases} \epsilon & \text{if } x = y\,,\\ \emptyset & \text{otherwise,} \end{cases} \tag{3.13}$$

$$d_x(R+S) = d_x(R) + d_x(S)\,, \tag{3.14}$$

$$\begin{aligned} d_x(R \cdot S) = d_x(R) \cdot S \\ + C(R) \cdot d_x(S)\,, \end{aligned} \tag{3.15}$$

$$d_x(R*) = d_x(R) \cdot R*\,, \tag{3.16}$$

$$d_x(Fork(R)) = Fork(d_x(R))\,. \tag{3.17}$$

Figure 3.2: Derivatives of FREs [4]

Derivatives of FREs are also slightly different than in (1.9) as seen in figure 3.2. The most important change is in the derivative of concatenation.

*Concurrent part* replaces null function. In a fork-free expression *concurrent part* gives the same result as null function, therefore *concurrent part* can be seen as null function lifted to FREs. It enables *Fork* prefixes to be ignored, therefore it is the pivot point in the concurrency of FREs. As a matter of fact the derivative of concatenation $R \cdot S$ considers two alternatives:

1. an event from $R$ occurs,
2. $R$ is a concurrent task and an event from $S$ occurs.

In the case that $R$ has a sequential part that cannot be skipped,

$$C(R) = \emptyset\,,$$

and the latter will not be considered. The explanation is that every leading *Fork* represents an already created concurrent task and as such it can execute anytime.

It is noticeable in (3.15), that the expression tries to describe all possibilities that can happen using alternation. This is the same approach as Taylor in [9]. An expression of program is similar in meaning to a *concurrency state* with its successors being its derivatives. This exploration of every state (possible or not) leads to *combinatorial explosion*

**Theorem 1** (Left quotient)**.** *[4] Let R be an Forkable Regular Expression and x be a symbol, then*

$$L(d_x(R)) = x \setminus L(R)\,. \tag{3.18}$$

This theorem proves that the derivatives correspond with trace language.

**Definition 17.** Let $R$ be an expression with defined language $L(R)$, then

$$\delta'(R) = \begin{cases} \{\epsilon\} & \epsilon \in L(R)\,, \\ \emptyset & \text{otherwise} \end{cases} \tag{3.19}$$

is a generalization of null function $\delta$ from definition 12.

**Theorem 2** (Representation)**.** *[4] Let R be a FRE, then*

$$L(R) = \delta'(R) \cup \bigcup_{x \in \Sigma} x \cdot L(d_x(R))\,. \tag{3.20}$$

Brzozowski in [3] showed that every regular expression can be written as a sum of its derivatives. That is exactly what theorem 2 is showing for Forkable Regular Expression.

**Definition 18** (Descendants)**.** [4] A descendant $S$ of a Forkable Regular Expression $R$ is either $R$ itself, a derivative of $R$, or the derivative of a descendant.

$$
\begin{aligned}
&R \simeq R && (Reflexivity) && (3.21)\\
&R \simeq S \land S \simeq T \implies R \simeq T && (Transitivity) && (3.22)\\
&R \simeq S \implies S \simeq R && (Symmetry) && (3.23)\\
&R + (S + T) \simeq (R + S) + T && (Associativity) && (3.24)\\
&R + S \simeq S + R && (Commutativity) && (3.25)\\
&R + R \simeq R && (Idempotence) && (3.26)\\
&R + \emptyset \simeq R && (Unit) && (3.27)\\
&\epsilon \cdot R \simeq R \cdot \epsilon \simeq R, \epsilon* \simeq \epsilon, Fork(\epsilon) \simeq \epsilon && (EmptyWord) && (3.28)\\
&\emptyset \cdot R \simeq R \cdot \emptyset \simeq \emptyset, \emptyset* \simeq \epsilon, Fork(\emptyset) \simeq \emptyset && (EmptyLanguage) && (3.29)\\
&E ::= [] \mid E* \mid R \cdot E \mid E + S \mid R + E \mid Fork(E) && (RegularContext) && (3.30)\\
&S \simeq T \implies E[S] \simeq E[T] && (Compatibility) && (3.31)
\end{aligned}
$$

Figure 3.3: Rules and axioms for similarity of FREs [4]

**Definition 19** (Similarity)**.** [4] Forkable Regular Expression $R$ and $S$ are similar, if $R \simeq S$ is derivable using rules and axioms from figure 3.3.

In the compatibility rule in figure 3.3 $E$ denotes a regular context. According to its grammar it has a hole inside denoted by $[]$. The notation $E[R]$ describes placing $R$ into this hole. Similarity is an equivalence relation, because it is reflexive transitive and symmetric.

The expression $Fork(R)*$ – called *iterated fork* – for $R = a \cdot b \cdot c$ describes a non-regular language and therefore can no longer be transformed into a finite automaton. Notice that in the definition (3.1) the creation of *iterated fork* is not restricted in any way.

**Definition 20** (Well-behaved)**.** [4] A FRE is well-behaved if all subterms of the form $R*$ have the property

$$C(d_w(r)) \leq \epsilon\,, \quad \forall w \in \Sigma^*\,. \tag{3.32}$$

Sulzmann and Thiemann in [4] proved that a well-behaved Forkable Regular Expression describe a regular language and therefore it is possible to transform into a finite automaton.

**Definition 21** (Dissimilar Descendants)**.** [4] The set of dissimilar descendants of $R$, $d_{\simeq}(R)$, is defined as a complete set of arbitrarily chosen representative FRE for the equivalence classes $d(R)/(\simeq)$.

**Theorem 3** (Finiteness of Well-Behaved Dissimilar Descendants)**.** *[4] Let $T$ be a well-behaved Forkable Regular Expression, then*

$$n(d_{\simeq}(T)) < \infty\,, \tag{3.33}$$

*where $n(A)$ is the cardinality of set $A$, $d_{\simeq}(T)$ is a set of dissimilar descendants of $T$.*

This proves, that the transformation will finish, because there is only a finite number of dissimilar descendants of a Forkable Regular Expression.

To find them a normal form of Forkable Regular Expression was proposed seen at figure 3.4.

$$R \oplus S = \begin{cases} R & R = S \vee S = \emptyset \\ S & R = \emptyset \\ R' \oplus (R'' \oplus S) & R = R' + R'' \\ (R \oplus S') + S'' & R \neq R' + R'', S = S' + S'', R < S' \\ S' + (R \oplus S'') & R \neq R' + R'', S = S' + S'', R \geq S' \\ R + S & R \neq R' + R'', S \neq S' + S'', R < S' \\ S + R & R \neq R' + R'', S \neq S' + S'', R > S' \end{cases}$$

$$R \odot S = \begin{cases} \emptyset & R = \emptyset \vee S = \emptyset \\ R & S = \epsilon \\ S & R = \epsilon \\ R' \odot (R'' \odot S) & R = R' \cdot R'' \\ S \cdot R & R = Fork(R'), S = Fork(S'), S' < R' \\ Fork(S') \cdot (R \odot S'') & R = Fork(R'), S = Fork(S') \cdot S'', S' < R' \\ R \cdot S & R \neq R' \cdot R'' \end{cases}$$

$$R^{\circledast} = \begin{cases} \epsilon & R = \emptyset \vee R = \epsilon \\ R* & otherwise \end{cases}$$

$$F(R) = \begin{cases} \emptyset & R = \emptyset \\ \epsilon & R = \epsilon \\ Fork(R) & otherwise \end{cases}$$

Figure 3.4: Normal form of Forkable Regular Expression [4]

## 3.2 New operators

The most defining feature of parallel programs is the creation of threads. That is what creates concurrency and with it non-deterministic behavior known as race conditions. FREs have *Fork* to model the effect of thread creation, thus it is possible to use FREs to search for race conditions.

The following subsections informally describe desired behavior of each operator and the motivation behind it.

### 3.2.1 Operator Atomic

In parallel programs, for the purpose of eliminating race conditions, synchronization measures are introduced, most notably locks. Locks make a section of code or a resource accessible only to one thread. The incentive is to disable all possibly intrusive elements when executing that section. Thus locks can be generalized into atomization of the section.

FREs themselves have no means to create an atomic section, therefore the operator *Atomic* can be introduced. *Atomic* describes atomic execution of a section of code. For language it would mean a restriction to interleaving as it would make a whole string appear as a single symbol. FREs extended by the operator *Atomic* are able to recognize deadlocks caused by the use of locks with a fair share of false positives caused by the generalization. Use of *Atomic* for truly atomic sections obviously does not give false positives.

It is intriguing to think about the scope of *Atomic*. A global atomic section makes sense when the expression is describing a run on a single computer. When distributed systems are taken into consideration, the scope of *Atomic* starts to matter. A local version of *Atomic* is more universal and thus is chosen for further consideration.

### 3.2.2 Operator Sync

One of the features of parallel programs is synchronization in the sense that the main thread or process waits for others to finish, enforcing their termination. Although FREs can model behavior of creating concurrent threads and processes, they cannot model this enforcement of termination. FRE waits for all concurrent tasks to finish only at the end of the expression. This is not enough as this does not support modularity, because a concatenation of two FREs shifts this end of expression. That is undoubtedly the desired behavior and it highlights the benefit of having a specialized operator.

The operator *Sync* (from synchronous) can be introduced to specify a scope for created threads. These threads run synchronously and are forced to terminate inside the scope, thus modeling the behavior of waiting. It can also be used as a means of sequencing parallel programs one after another.

*Sync* can also serve as a scope for local variant of *Atomic*. In the modeling of distributed system *Sync* would represent a single computer.

*Sync* is basically an inverted operator for *Fork* and, because of using it as a scope for *Atomic*, it is also an inversion of *Atomic*.

### 3.2.3 Operator Async

Asynchronous directly translates to "*not occuring at the same time*" or "*having each operation started only after the preceding operation is completed*". [15] Sometimes a set of unrelated tasks has to be executed. In this case the order of execution is not important. Considering the latter definition of asynchronous, the tasks can be executed one by one on a single thread. This behavior can be observed in simple implementations of callback programming or in the implementation of coroutines in Unity [16].

Operator *Async* describes this behavior. *Async* randomly sequences its operands without creating real concurrency. It is possible to model *Async* using *Fork*, *Atomic* and *Sync*.

## 3.3 Formal definition of new operators

**Definition 22.** Trace language of Extended Forkable Regular Expression (EFRE) is taken from definition 15 of trace language with following additions:

$$L(R) = deatom(L(R, \{\epsilon\}) \,, \tag{3.34}$$

$$L(Sync(R), K) = deatom(L(R)) \cdot K \,, \tag{3.35}$$

$$L(Atomic(R), K) = atom(L(R)) \cdot K \,, \tag{3.36}$$

where (3.34) replaces (3.2), $atom(U)$ of a language $U$ is treated as a symbol by operators and $deatom(V)$ recursively replaces all occurrences of $atom(U)$ in a language $V$ with $U$, using string operations lifted to sets.

This trace language of EFRE corresponds with the proposed operators in section 3.2.

**Definition 23.** Concurrent and sequential parts of Atomic and Sync are defined, using the generalized null function $\delta'$ from definition 17, in this manner:

$$C(Atomic(R)) = \delta'(R) \,, \tag{3.37}$$

$$C(Sync(R)) = \delta'(R) \,, \tag{3.38}$$

$$S(Atomic(R)) = R \,, \tag{3.39}$$

$$S(Sync(R)) = R \,. \tag{3.40}$$

To implement transformation using Brzozowski derivatives, expressions must have defined derivatives. The first goal of this part of thesis is to define rules for creating derivatives of the new operators and possibly even replace some of the old ones.

Every following subsection describes different approach to this problem, with the last one being satisfactory enough for implementation.

### 3.3.1 Using simple recursive function

The first notion was to use the rules of creating derivatives of FREs from figure 3.2 and define simple supporting recursive function, that would change the expression into another with an easy to find derivative.

It can be noticed that if a part of expression gets in front of *Fork*, it stops being part of its continuation language, therefore effectively stopping any interleaving. Function $\alpha(R)$ was defined to draw out *Atomic* parts from expression $R$ as seen in figure 3.5 and function $\beta$ to get the remaining expression without *Atomic* parts.

$$\alpha(Atomic(R)) = R \tag{3.41}$$
$$\alpha(R + S) = \alpha(R) + \alpha(S) \tag{3.42}$$
$$\alpha(R \cdot S) = \alpha(R) + C(R) \cdot \alpha(S) \tag{3.43}$$
$$\alpha(Fork(R)) = \alpha(R) \tag{3.44}$$
$$\alpha(x) = \epsilon \tag{3.45}$$

Figure 3.5: $\alpha$ function

Because skipping *Forks* prefixes happen during derivation of concatenation, the plan was to modify this rule so that the derivative of *Atomic* would precede all *Forks*. Derivation of expression $R \cdot S$ with respect to symbol $x \in \Sigma$ using function $\alpha$ and $\beta$ looks like this:

$$d_x(R \cdot S) = d_x(R) \cdot S + C(R) \cdot d_x(S) + d_x(\alpha(R)) \cdot \beta(R) \cdot S + d_x(\alpha(S)) \cdot C(r) \cdot \beta(S)\,. \tag{3.46}$$

The important part was for *Atomic* to have no derivatives on its own, i. e. for every symbol $x \in \Sigma$, $d_x(Atomic(R)) = \emptyset$.

Unfortunately, this approach does not work with branching in expressions. Counter example expression of $Fork(a)(Atomic(cd)e + Atomic(gh)i)$ is prepared into expression $(cd + gh)Fork(a)(e + i)$. This expression would describe string *cdai* but obviously it should not. The problem with extracting *Atomic* parts is visualized in figure 3.6.

### 3.3.2 Using very complex recursive functions

Using figure 3.6 as inspiration, it seems that instead of extracting *Atomic*, pushing *Fork* to after *Atomic* might work. This comes with the problem of duplicating parts of expression.

Another approach takes derivatives the expression processed by DeAtomic function on figure 3.8, that pushes concurrent behavior after the respective *Atomic* if possible. It is sufficient to call DeAtomic once, therefore derivation function $d_x()$ is replaced by a placeholder $d'_x()$, that calls it on the result of DeAtomic:

$$d'_x(R) = d_x(DeAtomic(R, \epsilon)) \tag{3.75}$$

The $\epsilon$ as second parameter $F$ symbolizes that no concurrent behavior is occuring at the beginning.

InsertDepth is a fairly simple function that is used inside DeAtomic to insert expressions into expressions. Full behavior of InsertDepth is described on

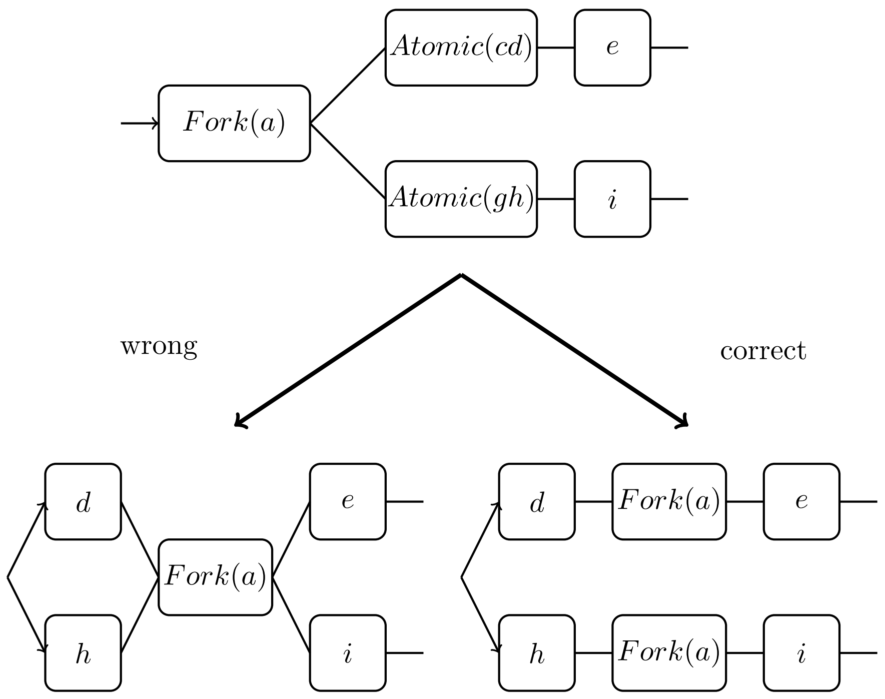

Figure 3.6: Flow of expression $Fork(a)(Atomic(cd)e + Atomic(gh)i)$

Let $R$, $T$ be EFREs and $n \in \mathbb{N}$ a number:

$$\begin{aligned}
InsertDepth(R, T, 0) &= T \cdot R\,, &(3.47)\\
InsertDepth(\emptyset, T, n) &= \emptyset\,, &(3.48)\\
InsertDepth(\epsilon, T, n) &= T\,, &(3.49)\\
InsertDepth(x, T, n) &= T \cdot x\,, &(3.50)\\
InsertDepth(R + S, T, n) &= InsertDepth(R, T, n)\\
&\quad + InsertDepth(S, T, n)\,, &(3.51)\\
InsertDepth(R \cdot S, T, n) &= InsertDepth(R, T, n) \cdot S\,, &(3.52)\\
InsertDepth(R*, T, n) &= (InsertDepth(R, T, n-1))*\,, &(3.53)\\
InsertDepth(Fork(R), T, n) &= Fork(InsertDepth(R, T, n-1))\,, &(3.54)\\
InsertDepth(Atomic(R), T, n) &= Atomic(InsertDepth(R, T, n-1))\,, &(3.55)\\
InsertDepth(Sync(R), T, n) &= Sync(InsertDepth(R, T, n-1))\,. &(3.56)
\end{aligned}$$

Figure 3.7: InsertDepth function

figure 3.7. Its first parameter is an expression into which should second param-

Let $R$, $S$ and $F$ be EFREs, $x \in \Sigma$ a symbol:

$$\begin{aligned}
\mathit{DeAtomic}(\emptyset, F) &= \emptyset\,, &(3.57)\\
\mathit{DeAtomic}(\epsilon, F) &= F\,, &(3.58)\\
\mathit{DeAtomic}(x, F) &= F \cdot x\,, &(3.59)\\
\mathit{DeAtomic}(R + S, F) &= \mathit{DeAtomic}(R, F) + \mathit{DeAtomic}(S, F)\,, &(3.60)\\
\mathit{DeAtomic}(R \cdot S, F) &= \mathit{DeAtomic}(R, F) \cdot S + \mathit{DeAtomic}(S, F \cdot C(R))\,, &(3.61)\\
\mathit{DeAtomic}(R*, F) &= \mathit{DeAtomic}(R, F) \cdot R * + F\,, &(3.62)\\
\mathit{DeAtomic}(\mathit{Atomic}(R), F) &= \mathit{DeAtomic}(R, \epsilon) \cdot F\,, &(3.63)\\
\mathit{DeAtomic}(\mathit{Fork}(R), F) &= \mathit{DeAtomicF}(R, \mathit{Fork}(\epsilon), 1) \cdot F\,, &(3.64)\\
\mathit{DeAtomic}(\mathit{Sync}(R), F) &= F \cdot \mathit{Sync}(\mathit{DeAtomic}(R, \epsilon))\,, &(3.65)
\end{aligned}$$

$$\begin{aligned}
\mathit{DeAtomicF}(\emptyset, F, n) &= \emptyset\,, &(3.66)\\
\mathit{DeAtomicF}(\epsilon, F, n) &= F\,, &(3.67)\\
\mathit{DeAtomicF}(x, F, n) &= \mathit{InsertDepth}(F, x, n)\,, &(3.68)\\
\mathit{DeAtomicF}(R + S, F, n) &= \mathit{DeAtomicF}(R, F, n) + \mathit{DeAtomicF}(S, F, n)\,, &(3.69)\\
\mathit{DeAtomicF}(R \cdot S, F, n) &= \mathit{DeAtomicF}(R, \mathit{InsertDepth}(F, S, n)) &\\
&\quad + \mathit{DeAtomicF}(S, F \cdot C(R), n)\,, &(3.70)\\
\mathit{DeAtomicF}(R*, F, n) &= \mathit{DeAtomicF}(R, \mathit{InsertDepth}(F, R*, n)) + F\,, &(3.71)\\
\mathit{DeAtomicF}(\mathit{Atomic}(R), F, n) &= \mathit{DeAtomic}(R, \epsilon) \cdot F\,, &(3.72)\\
\mathit{DeAtomicF}(\mathit{Fork}(R), F, n) &= \mathit{DeAtomicF}(R, \mathit{InsertDepth}(F, \mathit{Fork}(\epsilon), n), n + 1)\,, &\\
& &(3.73)\\
\mathit{DeAtomicF}(\mathit{Sync}(R), F, n) &= \mathit{InsertDepth}(F, \mathit{Sync}(\mathit{DeAtomic}(R, \epsilon)), n)\,. &(3.74)
\end{aligned}$$

Figure 3.8: DeAtomic function

eter be inserted. Third parameter specifies depth of insertion, i. e. into how many *Forks* and *Syncs* should the second parameter be nested in. Nonetheless, it is always placed at the beginning of an expression.

DeAtomic function is described on figure 3.8. It recursively scans reachable parts of expression collecting all concurrent behavior into expression $F$. Then placing it depending on the context. It has two modes:

- DeAtomic with the context set in a sequential part. When it comes across a symbol, it puts all concurrent behaviors in $F$ before the symbol, so that the symbol can be interleaved with it. When it comes across the operator *Atomic*, it places $F$ after it, effectively making the operator *Atomic* first in its scope and thus any unwanted interleaving is impossible.

- DeAtomicF with the context set in a concurrent part. Because any reached symbol is in the concurrent part, it is inserted into $F$. *Atomic* is still placed at the beginning of a scope outside of any *Fork*.

DeAtomic is clearly behaving differently in sequential context and concurrent context. This corresponds to switching contexts between *intra-thread* and *inter-thread*, called *bi-modal reasoning* by Sinha and Wang in [10].

This solution is neither effective nor elegant.

### 3.3.3 Exploiting concurrency

This approach considers sequential to be concurrent. If everything is inside *Fork*, then we do not need a special case for *Forks*. However, the problem with duplicating parts of expression persists.

**Definition 24.** The derivative of Extended Forkable Regular Expression $R$ with respect to some symbol $x \in \Sigma$ is defined inductively as follows:

$$\begin{aligned}
d_x(R) &= d_x(R, \epsilon)\,, && (3.76)\\
d_x(\emptyset, T) &= \emptyset\,, && (3.77)\\
d_x(\epsilon, T) &= \emptyset\,, && (3.78)\\
d_x(y, T) &= \begin{cases} Fork(T) & \text{if } x = y\,,\\ \emptyset & \text{otherwise}\,, \end{cases} && (3.79)\\
d_x(R+S, T) &= d_x(R, T) + d_x(S, T)\,, && (3.80)\\
d_x(R \cdot S, T) &= d_x(R, S \cdot T) + d_x(S, T) \cdot C(R)\,, && (3.81)\\
d_x(R*, T) &= d_x(R \cdot R*, T)\,, && (3.82)\\
d_x(Fork(R), T) &= d_x(R, \epsilon) \cdot Fork(T)\,, && (3.83)\\
d_x(Sync(R), T) &= Fork(Sync(d_x(R, \epsilon)) \cdot T)\,, && (3.84)\\
d_x(Atomic(R), T) &= Atomic(d_x(R, \epsilon)) \cdot Fork(T)\,. && (3.85)
\end{aligned}$$

The second parameter $T$ contains *sequential part* of continuation language. Basically it must not split operators, such as *Fork* or Sync, into two parts, so the context is held in $T$, until it can be all put into a single *Fork*.

The *combinatorial explosion* of FRE is still present with (3.81).

Extended Forkable Regular Expression using operators *Fork*, *Atomic* and *Sync* can model a fair number of other operators:

- *Async* doesn't bring any new expressivity as it can be singlehandedly built from the three:

$$Async(R, S, \dots) = Sync(Fork(Atomic(R)) \cdot Fork(Atomic(S)) \cdots)\,, \tag{3.86}$$

- Shuff function that shuffles contained symbols:

$$Shuff(R \cdot S \cdots) = Sync(Fork(Shuff(R)) \cdot Fork(Shuff(S)) \cdots)\,. \tag{3.87}$$

## 3.4 Proof of correctness

**Lemma 2.**

$$Fork(Fork(R) \cdot S) \equiv Fork(R) \cdot Fork(S)\,, \quad \textit{for every EFRE } R\,, S\,. \tag{3.88}$$

*Proof.* From definition 16:

$$\begin{aligned} &Fork(Fork(R) \cdot S) \equiv Fork(R) \cdot Fork(S) \\ &\quad \Longleftarrow L(Fork(Fork(R) \cdot S), K) = L(Fork(R) \cdot Fork(S), K)\,, \quad \forall K \in \Sigma^* \end{aligned}$$

Using associativity from Lemma 1 and definition 22 of trace languages:

$$\begin{aligned} L(Fork(Fork(R) \cdot S), K) &= L(Fork(R) \cdot S, \{\epsilon\}) \parallel K \\ \ldots &= L(Fork(R), L(S, \{\epsilon\})) \parallel K \\ \ldots &= L(R) \parallel L(S) \parallel K \\ \ldots &= L(R) \parallel L(Fork(S), K) \\ \ldots &= L(Fork(R), L(Fork(S), K) \\ \ldots &= L(Fork(R) \cdot Fork(S)), K) \end{aligned} \tag{3.89}$$

□

Lemma 2 allows flattening of *Forks*. During creation of derivatives, this is used at *Fork* rule as an example.

**Lemma 3.**

$$L(Fork(R)) = L(R)\,, \quad \textit{for every EFRE } R\,. \tag{3.90}$$

*Proof.* Using associativity from Lemma 1 and definition 22 of trace languages:

$$\begin{aligned} L(Fork(R)) &= L(Fork(R), \{\epsilon\}) \\ \ldots &= L(R) \parallel \{\epsilon\} \\ \ldots &= L(R)\,. \end{aligned} \tag{3.91}$$

□

Lemma 3 allows to completely disregard any sequential context, because the whole expression can be put into *Fork* before the creation of derivative.

**Lemma 4.** *[4]*

$$Fork(R) \cdot Fork(S) \equiv Fork(S) \cdot Fork(R)\,, \quad \textit{for every EFRE } R\,, S\,. \tag{3.92}$$

*Proof.* From definition 16:

$$\begin{aligned}&Fork(Fork(R)\cdot S) \equiv Fork(R)\cdot Fork(S)\\ &\quad\Longleftarrow L(Fork(Fork(R)\cdot S), K) = L(Fork(R)\cdot Fork(S), K)\,, \quad \forall K \in \Sigma^*\end{aligned}$$

Using associativity and commutativity from Lemma 1 and definition 15 of trace languages:

$$\begin{aligned}L(Fork(R)\cdot Fork(S), K) &= L(Fork(R), L(Fork(S), K))\\ \ldots &= L(R) \parallel L(Fork(S), K))\\ \ldots &= L(R) \parallel L(S) \parallel K\\ \ldots &= L(S) \parallel L(R) \parallel K\\ \ldots &= L(S) \parallel L(Fork(R), K)\\ \ldots &= L(Fork(S), L(Fork(R), K))\\ \ldots &= L(Fork(S)\cdot Fork(R), K)\,.\end{aligned} \tag{3.93}$$

□

Lemma 4 is used whenever a *Fork* is created. It is put at the back, but thanks to this lemma it makes no difference, because everything is in some kind of *Fork*.

**Conjecture 1** (Left quotient)**.** *Let $R$ be an Extended Forkable Regular Expression and $x$ be a symbol, then*

$$L(d_x(R)) = x \setminus L(R)\,. \tag{3.94}$$

**Conjecture 2** (Correctness)**.** *Theorems 2 and 3 still hold for Extended Forkable Regular Expression.*

Testing shows, that conjectures 1 and 2 are true, albeit a solid proof is still needed.

Chapter 4

# Implementation

This chapter documents important steps taken during the implementation of the theory, that was established in this thesis.

This implementation is mostly a proof of concept. Priority is given to comprehensibility, modularity and simplicity. Although not applicable all the time, object-oriented design was preferred.

Symbol alphabet consists of java Characters. This was chosen to keep the implementation as simple and comprehensible as possible. Symbol alphabet can contain any object as long as they implement total ordering, which is needed for normalization. However, when talking about regular expressions, mostly letters are expected to form an alphabet, strings and languages. It also makes the use of built-in classes `Pattern` and `Matcher` at least partially possible, but more about that will be disclosed in section 4.4.

In the following sections, each component of the whole implementation is described in greater detail.

## 4.1 Implementation of expression

FREs are implemented into Abstract Syntax Tree (AST). This structure shows very clearly how all the recursive functions work. Also the implementation of *normal form* from figure 3.4 is very transparent in order to show the meaning behind it.

FRE takes the form of a node in abstract syntactic tree. This node takes the form of the abstract class `FRExpression`. `FRExpression` extends the abstract class `RegularExpression`, which consists of `derivateBy` and `normalize` methods and `isEmpty` and `isNullable` tests. `FRExpression` then adds concurrent part C and sequential part S. On top of that, it has methods `stdDerivateBy`, that corresponds to the native derivative of FRE, and `extDerivateBy`, that was devised in this thesis as a part of EFRE.

`FRExpression` also implements an interface `Comparable<FRExpression>`, because one of the conditions of *normal form* is the total ordering of expressions. It has these abstract subclasses:

- `FRConstant`,
- `FRUnaryOperator`,
- `FRBinaryOperator`.

A special class named `FRConstant` was created to represent leaves of the AST. It is always in normal form. Subclasses of `FRConstant` are:

- `FREmpty`, representing $\emptyset$,
- `FRNull`, representing $\epsilon$,
- `FRLiteral`, representing symbol $x \in \Sigma$.

Class `FRUnaryOperator` represents a function or unary operator. It contains methods to work with a single operand. In the AST it has exactly one child. It represents superclass for:

- `FRStar` as Kleene star operator,
- `FRFork` as *Fork* operator,
- `FRAtomic` as *Atomic* operator,
- `FRSync` as *Sync* operator.

Class `FRBinaryOperator` represents a binary operator, defines methods to work with left and right operands and as expected has two children in the AST, which are its operands. It has subclasses:

- `FRConcatenation`,
- `FRAlternation`.

Method normalize is based on normal form on figure 3.4.

There is no class for the representation of the operator *Async* because it can be substituted by a clever use of *Fork*, *Atomic* and *Sync*.

## 4.2 Basic automaton implementation

Interface `FiniteStateMachine`, representing a deterministic finite automata, exactly copies definition 2 of finite automata restricted by definition 5.

There are multiple variants to implement this interface. Simple notation variants given by [1] are:

- a transition diagram, that is a graph,
- a transition table, consisting of states denoting rows and symbols of the alphabet denoting columns, in a cell in a row $q$ and a column $x$ there is a state $q'$, such that $\delta(q, x) = q'$.

The table can be implemented as integer matrix $|Q| \times |\Sigma|$ with direct mapping of integers on states and on symbols. Final states can be recognized by an array of boolean flags and the starting state can be mapped onto 0. It is apparent, that non-deterministic behavior cannot happen. Required memory space is then

$$\begin{gathered} O(|Q| * |\Sigma|)\,, \\ \text{assuming } |\Sigma| \ll |Q|\,, \text{ it is } O(|Q|)\,. \end{gathered} \tag{4.1}$$

Required time for finding a transition is the same finding in random access to a matrix and that is in constant time:

$$O(1)\,. \tag{4.2}$$

Class `BasicFiniteAutomata` implements the graph variant. It holds a set of nodes *states*, equivalent with a set of states $Q$, a set of input symbols *alphabet*, a set of edges *edges*, that use a pair of $(q, x)$, $q \in \mathit{states}$, $x \in \mathit{alphabet}$ as a key. This fulfills the condition of being a deterministic finite automaton. It also remebers the starting state *startState* and the set of final states *endStates*. Required memory space is

$$\begin{aligned} &O(|Q| + |\Sigma| + |Q| * |\Sigma|) = O(|Q| * |\Sigma|)\,, \\ &\text{assuming } |\Sigma| \ll |Q|\,, \text{ it is } O(|Q|)\,. \end{aligned} \tag{4.3}$$

Required time for finding a transition is the same as finding an element in a set and that is

$$O(\log(|Q|))\,. \tag{4.4}$$

The graph was chosen, despite worse performance, for its clarity. Comprehensibility – as stated at the beginning of this chapter – has bigger significance than running speed.

Class `BasicFiniteAutomata` is just a skeleton and it needs class `FSMResolver` to simulate its run. `FSMResolver` was designed so that it can simulate any automaton-class implementing `FiniteStateMachine` interface.

It knows a current state, makes transitions using method `makeStep` and can answer a simple query, whether the automaton is in an accepting configuration or not. It is very lightweight and the automaton-class can be shared among many `FSMResolvers`.

## 4.3 Method of transformation

Class `BasicFiniteAutomatonBuilder` is responsible for the transformation of FRE into finite automaton. From defined input alphabet and expression to transform it creates `BasicFiniteAutomaton` using the method of Brzozowski derivatives adapted from [3]. The algorithm goes as described on figure 4.1:

## 4.4 Pattern and Matcher

FREs extend regular expressions.

Special variation of regular expressions is already implemented in java. In java, class Pattern is used to represent an expression. Class `Matcher` is then created to match a specified input against this `Pattern`. Those classes are marked as final, which means that a subclass for FRE cannot be created. [17, 18]

However, those classes can be used as an inspiration for `FRPattern` and `FRMatcher`. `FRPattern` implements the most fundamental methods of `Pattern`, but with the use of FRE. It can compile strings of lowercase letters and symbol operators. Special labels, such as *fork*, *sync*, *atomic*, *async* and *null* for $\epsilon$ and *empty* for $\emptyset$, have to be preceded by quote character (by default /). `FRMatcher` gives more control over matching such as specifying parts of input string to be matched and resetting matching progress.

## 4.5 Parser

A fundamental ability of Pattern is compiling regular expressions from strings. For `FRPattern` to have similar functionality, it has to be able to parse FREs. Fortunately, FREs have very basic syntax.

There are two basic outlooks on parsing: top-down and bottom-up. Assuming the parsing is done into AST, top-down starts with the root and builds its way down. It struggles with left recursion, because it does not know, where the root will be. On the other hand, bottom-up starts from small pieces and joins them together, building the tree from the bottom with root being the last node built. It can deal with left and right recursion, but it has a problem with identifying a wrong input. [19]

For the purpose of parsing FREs the shunting-yard algorithm was used. It is a specialized algorithm for parsing expressions (not necessarily regular expressions) based on the bottom-up principle. It uses two stacks, one for

```
Method: AddUnique
Input: FRE $R$
1. if $R$ is not in *derivatives*:
2.     add $R$ to *states*,
3.     add $R$ to *derivatives*,
4.     push $R$ to *queue*,
5.     if $R$ isNullable:
6.         add $R$ to *endStates*.
```

```
Algorithm: Brzozowski derivatives
Input: alphabet $\Sigma$, FRE $R$
Output: BasicFiniteAutomaton
1. *startState* is $R$,
2. AddUnique($R$),
3. while there are FREs in *queue*:
4.     $S$ = *queue* pop,
5.     for each symbol $x \in \Sigma$:
6.         create a derivative $D$ of $S$ with respect to $x$,
7.         AddUnique($D$),
8.         add $(S, x, D)$ to *edges*,
9. output BasicFiniteAutomaton(*states*, *edges*, $\Sigma$, *startState*, *endStates*).
```

Figure 4.1: Transformation algorithm [3]

operands and one for operators. Read operands are put onto the operands stack and operators on the operator stack, unless there are operators with higher precedence. If there are, they are combined with operands from the-operands stack and then pushed onto the operands stack. [20]

`FRExpressionBuilder` is using a slightly modified version of Shunting-yard algorithm with the capability to distinguish between left associative and right associative unary operators. This capability was essential, because FREs have Kleene star $*$ that is right associative and *Fork* that can be treated as left associative operator.

## 4.6 Testing

The functionality of project is tested using a couple of classes:

- `FRETest` tests `FRExpression` and its subclasses, it checks normalization rules, derivation rules and parsing FRE from strings.
- `BFATest` tests creation of `BasicFiniteAutomaton`, namely its states and edges, and correct simulation of FMSResolver.
- `FRPatternMatcherTest` confirms correct behavior of all the modules through a series of specially designed FREs.
- `AnalysisTest` shows how FREs can be used to find race conditions and some special examples of deadlocks.

### 4.6.1 Analysis example

First example is of a simple race condition on figure 4.2. The program can be translated into Forkable Regular Expression

$$Fork((a \cdot b \cdot c)*) \cdot Fork((a \cdot b \cdot c)*)\,.$$

Reading is $a$, incrementation is $b$ and writing is $c$. A Read after Write hazard can happen if another $a$ gets between $a$ and $c$.

```
/fork((abc)*)/fork((abc)*) <- abacbc : match
```

And the hazard was correctly recognized.

If locks are added to the program in order to remove this hazard, the expression changes to

$$Fork(Atomic(a \cdot b \cdot c)*) \cdot Fork(Atomic(a \cdot b \cdot c)*)\,.$$

The result changes to:

```
/fork(/atomic(abc)*)/fork(/atomic(abc)*) <- abacbc : not match
```

The hazard was successfully removed.

In case of deadlock in figure 4.3, it is needed to construct and expression describing the opposite of deadlock. Such expression is

$$Fork(Atomic(ab))Fork(Atomic(cd))\,.$$

```
public class TestRaceCondition implements Runnable {

    static int Counter;

    public void run() {
        for (int i = 0; i < 10000; ++i) {
            int tmp = Counter;
            tmp += 1;
            Counter = tmp;
        }
        System.out.println(Counter);
    }


    public static void main(String args[]) {
        Counter = 0;
        (new Thread(new TestRaceCondition())).start();
        (new Thread(new TestRaceCondition())).start();
    }
}
```

Figure 4.2: Example of race condition

Then the event output from the program is used to match against this expression. The matching goes step by step and announces when it stops matching. That shows a deadlock happened.

Considering example output of $ac$, it stops matching at $c$ and announces deadlock. With output of $cdab$, it matches, which means no deadlock happened.

It might be more apparent if the event is created inside the waiting loop. Output of such program would be infinite, but the deadlock would be recognized in just a few symbols.

The deadlock in figure 4.3 is called livelock, because it does not yield and wait for access. It makes very little difference to the example.

```
public class TestDeadLock implements Runnable {

    static AtomicBoolean A;
    static AtomicBoolean B;

    boolean inverted;

    void getAB()
    {
        while (!A.compareAndSet(false, true)) {}
        System.out.println("a");
        while (!B.compareAndSet(false, true)) {}
        System.out.println("b");
        B.set(false);
        A.set(false);
    }

    void getBA()
    {
        while (!B.compareAndSet(false, true)) {}
        System.out.println("c");
        while (!A.compareAndSet(false, true)) {}
        System.out.println("d");
        A.set(false);
        B.set(false);
    }

    @Override
    public void run() {
        if (inverted)
            getBA();
        else
            getAB();
    }

    TestDeadLock(boolean b) {
        inverted = b;
    }
    public static void main(String args[]) {
            A.set(false);
            B.set(false);
            (new Thread(new TestDeadLock(false))).start();
            (new Thread(new TestDeadLock(true))).start();
    }
}
```

Figure 4.3: Example of deadlock

# Outlook

Proving conjectures 1 and 2 is very important for the future progress. It can shed insight on the structure of synchronization events as introduced in this thesis.

Implementation provided by this thesis is not very scalable in terms of expression length. Nevertheless, it is substantially modular so adding new operators is easy. Assuming that FREs will be used in practice, another more optimized transformation method will be needed, because the method using Brzozowski derivatives is known to be slow.

It is also worth considering to create a separate operator DeAtomize as an inversion to *Atomic* instead of using *Sync* for this purpose.

Adding Boolean functions to FRE could yield interesting results. Especially AND function, because it makes matching of expression against expression possible. One FRE could describe a program and the other could describe a hazard. Then just checking the accessibility of final states of the resulting automaton would reveal the existence of the hazard in the program.

# Conclusion

The first goal was to add other operators to bring FRE closer to parallel programs. In chapter 3 operators *Atomic*, *Sync* and *Async* were selected and designed for this task.

The definition of the operator *Atomic* was the trickiest one. Upon closer inspection the meaning of *Atomic* forks into two possible semantically different operators: local atomic and global atomic. The local version of *Atomic* was chosen, because it can be used to model multiple layers of concurrency. This was intriguing as it corresponds with distributed computing.

The operator *Sync* was defined to represent a sequential string of literals, that is matched by the contained expression. This means that it does not exhibit any concurrent behavior itself even if its content does. Moreover it is the perfect operator for defining the scope of the local atomic.

The operator *Async* could be constructed using already defined operators (in this case atomic, sync and fork). This construction was preferred over a stand-alone definition, because of its simplicity and comprehensibility. The construction also revealed another upside of using local atomic. When local atomic was used as part of async it made interleaving of fork with asynchronous tasks possible.

Another goal was to implement a method to transform FRE with these added operators into a finite state machine using Brzozovsky derivatives.

For this task the class `FRExpressionBuilder` was created. An instance of `FRExpressionBuilder` parses FRE from string format and outputs an Abstract Syntax Tree of `FRExpression`s that is then processed into a finite state-machine `BasicFiniteAutomaton` by a `BasicFiniteAutomatonBuilder`. Any automaton-representing class that implements the `FiniteStateMachine` interface can be simulated using a `FSMResolver` and that is the last step of the process. Classes `FRPattern` and `FRMatcher` encapsulate this process and make it simple to use.

Testing showed that the rules for the new operators were developed correctly. Moreover the increased expressiveness did affect overall complexity less than by an anticipated margin. Speed was an issue, but that was expected due to the *combinatorial explosion* of states.

# Bibliography

[1] Hopcroft, J. E.; Motwani, R.; et al. Introduction to automata theory, languages, and computation. *Acm Sigact News*, volume 32, no. 1, 2001: pp. 60–65.

[2] Rozenberg, G.; Salomaa, A. *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer Science & Business Media, 2012.

[3] Brzozowski, J. A. Derivatives of regular expressions. *Journal of the ACM (JACM)*, volume 11, no. 4, 1964: pp. 481–494.

[4] Sulzmann, M.; Thiemann, P. Forkable regular expressions. In *International Conference on Language and Automata Theory and Applications*, Springer, 2016, pp. 194–206.

[5] Gramlich, G.; Schnitger, G. Minimizing nfa's and regular expressions. *Journal of Computer and System Sciences*, volume 73, no. 6, 2007: pp. 908–923.

[6] DuPaul, N. Static Testing vs. Dynamic Testing. Veracode [online], Updated: 18. 7. 2017 [cit. 17. 4. 2018]. Available from: `https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing`

[7] Young, M.; Taylor, R. N. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, volume 14, no. 10, 1988: pp. 1499–1511.

[8] Duesterwald, E.; Soffa, M. L. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*, ACM, 1991, pp. 36–48.

[9] Taylor, R. N. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, volume 26, no. 5, 1983: pp. 361–376.

[10] Sinha, N.; Wang, C. Staged concurrent program analysis. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ACM, 2010, pp. 47–56.

[11] Amtoft, T.; Nielson, F.; et al. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, volume 7, no. 3, 1997: pp. 321–347.

[12] Avrunin, G. S.; Dillon, L. K.; et al. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, volume SE-12, no. 2, 1986: pp. 278–292.

[13] Garg, V. K.; Ragunath, M. Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Science*, volume 96, no. 2, 1992: pp. 285–304.

[14] Leiß, H. Towards Kleene algebra with recursion. In *International Workshop on Computer Science Logic*, Springer, 1991, pp. 242–256.

[15] Dictionary.com Unabridged. Asynchronous. [online], May 2018 [cit. 9. 5. 2018]. Available from: `http://www.dictionary.com/browse/asynchronous`

[16] bunny86. Answer in Unity3D and C# - Coroutines vs threading. [online], Last update: November 2012 [cit. 9. 5. 2018]. Available from: `http://answers.unity.com/answers/357040/view.html`

[17] Oracle. Pattern (Java Platform SE 7). [online], 2018 [cit. 9. 5. 2018]. Available from: `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`

[18] Oracle. Matcher (Java Platform SE 7). [online], 2018 [cit. 9. 5. 2018]. Available from: `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html`

[19] Kegler, J. Parsing: Top-down versus bottom-up. [online], Last update: 15. 11. 2014 [cit. 10. 5. 2018]. Available from: `http://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2014/11/ll.html`

[20] Norvell, T. Parsing Expressions by Recursive Descent. [online], 1999 [cit. 9. 5. 2018]. Available from: `www.engr.mun.ca/~theo/Misc/exp_parsing.htm`

Appendix A

# Glossary

**call graph** Is a control flow graph displaying calls between procedures..

**combinatorial explosion** Is a quick increase in complexity because of combinatorics..

**concurrency state** Represents a status of tasks in a flow graph and a set of its successors is defined..

**concurrency history** Is a sequence of concurrency states..

**control flow graph** A flow graph describing possible paths of execution..

**halting problem** Is a decision problem of whether a program stops or runs forever..

**inter-thread** Is a concurrent context..

**intra-thread** Is a sequential context..

**lock** Is a synchronization construct used to restrict race conditions..

**NP-complete** Is a set of problems belonging to NP and NP-hard at the same time..

**NP-hard** Is a set of problems that it can be reduced to every NP problem in polynomial time..

**PSPACE** Is a set of decision problems solved by Turing machine using polynomial amount of space. $NP \subseteq PSPACE$..

**scope information** Characterizes where declarations still apply..

**symbolic execution** Is a method, that evaluates a program based on abstract inputs..

**task** Is a sequence of actions, creating a sequence of events..

Appendix B

# Acronyms

| | |
|---|---|
| **AST** | Abstract Syntax Tree. |
| **CRE** | Concurrent Regular Expression. |
| **EFRE** | Extended Forkable Regular Expression. |
| **FRE** | Forkable Regular Expression. |

Appendix C

# Contents of enclosed CD

```
readme.txt ......................... the file with CD contents description
src ........................................ the directory of source codes
text ........................................... the thesis text directory
    thesis ............... the directory of LaTeX source codes of the thesis
    thesis.pdf .......................... the thesis text in PDF format
```