NTNU

Norwegian University of
Science and Technology

Designing and computing bounds for non-deterministic state machines

Stepan Alexandrov

Embedded Computing Systems
Submission date: June 2016
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

NTNU
Norwegian University of
Science and Technology

Designing and Computing Bounds for Nondeterministic State Machines

Stepan Alexandrov

Submission date: June 2016
Responsible professor: Kjetil Svarstad, IET
Supervisor:

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

June 9, 2016

Contents

List of Figures

List of Tables

Chapter 1

Problem Description

A nondeterministic finite state acceptor (NFSA) can be described by a regular expression and vice versa. This can be used to implement regular expression matching in hardware with dynamic reconfiguration. Regular expression matching implemented this way proved to be feasible and shows good performance/flexibility relation [1]. When an NFSA is extended to allow outputs on each state transition, we will refer to it as a nondeterministic finite state machine (NFSM), it becomes possible to implement control logic using regular expressions plus information about outputs. The advantage of an NFSM over a deterministic finite state machine (DFSM) is that an NFSM can have much less states than an equivalent DFSM [2], but since execution of an NFSM involves copying of it on each nondeterministic state transition and its further parallel execution (an NFSM should execute faster than a DFSM), the number of copies can grow very fast. To execute an NFSM on a FPGA, we need to know maximum possible number of copies of the NFSM that can be created during execution. If the maximum number of copies is less or equal to the number supported by the FPGA, then we say that the NFSM is executable on the FPGA. The main task of this thesis is to develop a software program which will accept a regular expression, transform it to the NFSM, and compute the maximum number of NFSM copies (bound) which may be created when the NFSM is executed.

The subtasks include a research on NFSMs and DFSMs, adaptation and implementation of the Thompson's transformation algorithm, development and implementation of a bound computation algorithm, execution of an NFSM in software, visualization of an NFSM transformed by the Thompson's algorithm.

Chapter 2

Abstract

Dynamically reconfigurable hardware receives more and more attention these days. This is on no account by accident, but due to very useful characteristics, such as high flexibility and performance. In this work we consider dynamically reconfigurable hardware as means to execute an NFSM.

Execution of an NFSM is performed by making a copy of it on each non-deterministic state transition and executing all the copies in parallel. Parallel execution achieves high processing performance, but requires more hardware.

In this work a software program is developed which accepts a regular expression, transforms it to the corresponding NFSM, saves its structural representation in a file, computes the bound on copies, and executes the NFSM. The program has been tested with different types of regular expressions, and it showed to be correct in the transformation and the bound computation. For visualization of an NFSM diagram a free open source software GraphViz is used.

Chapter 3

Preface

Writing the master thesis was quite a great experience. Developing the software program, which is approximately 2500 lines of code in size, presented many challenges, but was very joyful experience. The program has been tested as much as time allowed, but absence of errors is not guarantied.

Many thanks goes to Prof. Kjetil Svarstad for guidance and support provided during writing of the master thesis. Also I would like to thank all those people who share their experience on Stackoverflow website stackoverflow.com.

Chapter 4

List of Abbreviations

DFSA	Deterministic Finite State Acceptor
DFSM	Deterministic Finite State Machine
NFSA	Nondeterministic Finite State Acceptor
NFSM	Nondeterministic Finite State Machine
FPGA	Field-programmable Gate Array
POSIX	Portable Operating System Interface for UNIX
BRE	Basic Regular Expressions
ERE	Extended Regular Expressions
MFC	Microsoft Foundation Classes
GUI	Graphical User Interface
MSDN	Microsoft Developer Network
STL	Standard Template Library
ID	Identifier
UI	User Interface

Chapter 5

Introduction

Regular expressions and equivalent to them NFSMs are extensively used in software applications ranging from compilers [3] to text editors and programming languages [4]. The usage of a regular expression is primarily to search for a set of substrings in a string. A "string" can be a text file, user input, Ethernet packets etc. Implementation of an NFSM on an FPGA in some cases can offer considerable speed-up of regular expression matching. This finds its use in Internet traffic monitoring, where processing speed is important. Another usage of NFSMs on an FPGA is meta-computations and control logic implementation.

5.1 Motivation

When an NFSM is executed, i.e. a regular expression matching is performed, it is copied on non-deterministic state transitions, so that copies represent different possibilities of a non-deterministic transition. A string is accepted if one of the NFSM copies reach the accept state. Since FPGAs have limited resources, it is required to know maximum number of copies of NFSM before it is executed. As an example, let us consider the FPGA depicted in figure 5.1. The FPGA implements an NFSM, which occupies certain amount of hardware resources. In our case, the FPGA supports 15 copies of the NFSM. The number of supported copies of an NFSM depends on the hardware resources of an FPGA and on the size of the NFSM. When the FPGA stars execution of the NFSM, it has only one NFSM implemented in hardware. As the execution progresses, the number of copies of the NFSM grows due to non-deterministic state transitions. This leads us to the conclusion that the FPGA reconfiguration logic should check that the NFSM will not exceed the limit on copies when it is in executing, no matter what input arrives. This check should occur before the NFSM is executed.

In this master thesis we aim, given a regular expression, transform it to the corresponding NFSM, compute the bound on copies, and execute it. Execution of an NFSM is performed in software in this work.

Figure 5.1: Example of an FPGA implementing an NFSM.

5.2 Objectives

The objectives of the thesis are following:

1. Perform research on properties of NFSM and DFSM, and their transformation algorithms.
2. Adapt and implement in software Thompson's transformation algorithm.
3. Develop and implement in software an algorithm for computation of bounds.

5.3 Contribution

The following have been achieved in the thesis:

1. Classification of NFSMs as having regular and irregular structure based on their construction algorithms.
2. Prove that the NFSMs constructed by the adapted Thompson's transformation algorithm together with the execution rules always have bound on copies independent of the string length to be matched.
3. Development of the bound computation algorithm.
4. Implementation in software and testing of the Thompson's transformation algorithm and the bound computation algorithm.

5.4 Method

During the work different approaches to solving the problems have been taken. In the case of proving the properties of NFSMs constructed by the adapted Thompson's transformation algorithm, the method based on induction is used, though the inductive step is not distinctively present due to the nature of the problem.

In the development of the bound computation algorithm, try-and-error method is followed, with initial solution based on inductive thinking. The try-and-error method is conducted based on software implementations of the bound computation algorithm. It took two iterations to reach the most accurate (as time allowed) bound computation algorithm. Performance is the second in importance.

5.5 Thesis Overview

During the work on the thesis, a great deal of effort has been spent on programming, debugging, and testing of the software program. The source code and the compiled program is supplied with the thesis. The Microsoft Visual Studio Community 2015 used for the development is available for free on the official website.

The chapter 2 presents the automata theory and related topics on sets and graphs. At the end of the chapter, execution of an NFSM is explained and adaptation of the Thompson's transformation algorithm is presented. In the chapter 3, an algorithm for computing bounds on number of NFSM copies, which are created when an NFSM executes, is developed. It is shown that NFSMs constructed by the adapted Thompson's algorithm have regular structure, and together with the execution rules, presented in the same chapter, significantly simplify the development of the bound computation algorithm. In fact, such regular NFSMs always have bound. Chapter 3 contains description of the program architecture. It documents classes and their members. At the end of the chapter, a class diagram and an object diagram can be found. The class digram shows composition and relations of the classes in the program to each other. The class diagram is static representation of the program structure. The object diagram is dynamic (run-time) representation of the program structure. It shows how objects instantiated from the classes interact with each other.

Chapter 6

Theory and Background

6.1 Mathematical Preliminaries and Notation

6.1.1 Sets

A set is a collection of elements without any structure other than membership. Expression $x \in S$ indicates that x is an element of the set S. To indicate that x is not an element of the set S, we write $x \notin S$. To specify a set, a description of its elements is enclosed in curly braces, for example

$$S = \{0, 1, 2, 3, ...\} = \mathbb{N} \tag{6.1.1}$$

(a set of all natural numbers). The description of a set of all even natural numbers is

$$S = \{x : x \geq 0, x \ is \ even\} \tag{6.1.2}$$

The *universal set* U is a set which contains all the elements (all natural numbers for example). An *empty set* $\emptyset$, is a set which contains no elements [5].

The following operations are defined for sets:

- *union* ($\cup$)
$$S_1 \cup S_2 = \{x : x \in S_1 \ or \ x \in S_2\} \tag{6.1.3}$$
- *intersection* ($\cap$)
$$S_1 \cap S_2 = \{x : x \in S_1 \ and \ x \in S_2\} \tag{6.1.4}$$
- *difference* (-)
$$S_1 - S_2 = \{x : x \in S_1 \ and \ x \notin S_2\} \tag{6.1.5}$$
- *complementation* ($\bar{S}$), if universal set U is given then
$$\bar{S} = \{x : x \in U \ and \ x \notin S\} \tag{6.1.6}$$

For example, U can be a set of all natural numbers and S is a set of all even natural numbers, then $\bar{S}$ is the set of all odd natural numbers. The following relations hold for sets:

$$\bar{\bar{S}} = S,\ \bar{\emptyset} = U,\ S \cap U = S,\ S \cap \emptyset = \emptyset,\ S \cup U = U,\ S \cup \emptyset = S \tag{6.1.7}$$

The Demorgan's laws [5]:

$$\overline{S_1 \cup S_2} = \overline{S_1} \cap \overline{S_2},\ \overline{S_1 \cap S_2} = \overline{S_1} \cup \overline{S_2} \tag{6.1.8}$$

A set S_1 is a *subset* of S_2, denoted $S_1 \subseteq S_2$, if every element of S_1 is also an element of S_2. If S_1 is a subset of S_2 and S_2 contains some elements not in S_1 , then S_1 is a *proper subset*, denoted $S_1 \subset S_2$. Two sets S_1 *and* S_2 are said to be *disjoint* if they have no elements in common ($S_1 \cap S_2 = \emptyset$) [5].

A set is *finite* if it contains a finite number of elements, otherwise it is *infinite*. The size of the finite set S equals to the number of its elements, denoted $\mid S \mid$. A set of all subsets of a set S is called a *power set* of the set S and is denoted 2^S. Empty set $\emptyset$ is included in a *power set* [5].

The Cartesian product of two sets is defined as [5]:

$$S = S_1 \times S_2 = \{(x, y) : x \in S_1, y \in S_2\}. \tag{6.1.9}$$

The Cartesian product of n sets is defined as [5]:

$$S = S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, ..., x_n) : x_i \in S_i\}. \tag{6.1.10}$$

6.1.2 Functions

A function f assigns a unique element from a set A to one or more elements of a set B, written $f : B \to A$, where B is the domain and A is the range of the function f, written $Domain(f)$, $Range(f)$. The function f is called *partial function* if $Domain(f) \subset B$, otherwise if $Domain(f) = B$, the function f is *total function* [5].

In some cases we are only interested in the growth rate of a function when its arguments become very large. Common order of magnitude notation is used to relate growth rate of one function to growth rate of another function. Considering two functions $f(n)$ and $\lambda(n)$ with $n \in \mathbb{N}$, the following growth rate relations possible [5]:

- $f(n)$ has order at most the order of $\lambda(n)$, denoted $f(n) = O(\lambda(n))$, if

$$\mid f(n) \mid \leq c \mid \lambda(n) \mid,$$

- $f(n)$ has order at least the order of $\lambda(n)$, denoted $f(n) = \Omega(\lambda(n))$, if

$$\mid f(n) \mid \geq c \mid \lambda(n) \mid,$$

- $f(n)$ has the same order as $\lambda(n)$, denoted $f(n) = \Theta(\lambda(n))$, if

$$c_1 \mid \lambda(n) \mid \leq \mid f(n) \mid \leq c_2 \mid \lambda(n) \mid,$$

  where c, c_1, and c_2 are constants.

In the order of magnitude notation, multiplicative constants and lower order terms are ignored, since they become negligible as n increases.

Relation is more general concept than function: relation can assign to an element from the domain many elements in the range, where as function assigns only one element in its range. *Equivalence*, which is a generalization of the concept of equality, is a relation. To indicate that x is equivalent to y, we write $x \equiv y$.

equivalence has the following properties [5]:

- *reflexivity*

$$\forall x : x \equiv x,$$

- *symmetry*

$$if\ x \equiv y\ then\ y \equiv x,$$

- *transitivity*

$$if\ x \equiv y\ and\ y \equiv z,\ then\ x \equiv z.$$

6.1.3 Graphs and trees

A *directed graph* is a construct defined by two sets: the set $V = \{v_1, v_2, ..., v_n\}$ of *vertices* and the set $E = \{e_1, e_2, ..., e_m\}$ of *edges*, where each edge is a pair of vertices from V. For example $e_i = (v_j, v_k)$ is an edge from v_j to v_k. Graphs are visualized by diagrams in which the vertices are represented as circles and the edges as lines with arrows connecting the circles [5]. For example, the directed graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_3, v_1), (v_2, v_3), (v_3, v_2), (v_2, v_2), (v_2, v_1)\}$ is depicted in the figure 6.1.

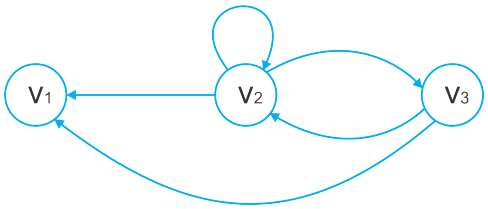


Figure 6.1: Example of a directed graph.

A *walk* from v_i to v_n is a sequence of edges $(v_i, v_j), (v_j, v_k), ..., (v_m, v_n)$. A length of a walk is the total number of edges of the walk. A *path* is a walk in which no edge is repeated. A *simple path* is a path in which no vertex is repeated. A *cycle* is a path form vertex v_i to itself, where v_i is the *base* of the

cycle. A *simple cycle* is a cycle in which no vertex is repeated except the base. A *loop* is an edge from a vertex to itself [5].

A *tree* is a directed graph without cycles and loops, it has a *root* which is a vertex, such that there is exactly one path from the root to every other vertex. A *leaf* of a tree is a vertex which has no outgoing edges. If there is an edge $e_i = (v_i, v_j)$ in a tree, then v_i is said to be the *parent* of v_j, and v_j is the *child* of v_i. The *level* of a vertex is he number of edges in the path from the root of the tree to the vertex. The *height* of a tree is the largest level among the vertices. An *ordered tree* is a tree where there is an ordering associated with the vertices at each level of the tree. An example of a tree is depicted in the figure 6.2 [5].

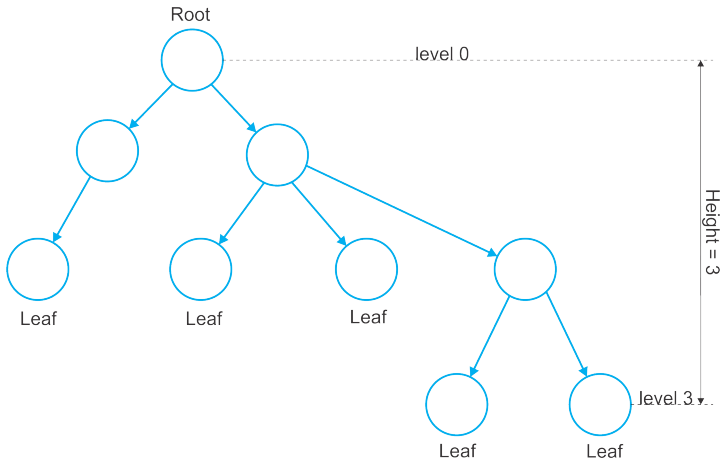


Figure 6.2: Example of a tree.

6.1.4 Formal languages

An *alphabet* is a finite, nonempty set Σ of symbols from which *strings* are formed. Strings are finite sequences of symbols from the alphabet. Let $\Sigma = \{r, m\}$, $s_1 = rmrmrr$, and $s_2 = mmmmmr$ be an alphabet, and two strings formed from that alphabet respectively, then *concatenation* of the two strings, written $s_1 s_2$, will be also a string:

$$s_1 s_2 = rmrmrrmmmmmr, \tag{6.1.11}$$

which is done by appending symbols from s_2 to the right end of the string s_1 [5].

The *reverse* of a string s_1 is a string s_1^R formed by writing the symbols of the string s_1 in reverse order: $s_1^R = rrmrmr$. The *length* of a string s, written $\mid s \mid$, is the number of symbols in the string. An *empty string* λ is a string with no elements. Concatenation of any string with the empty string has no effect. If s is a string, then s^n is a string obtained by concatenation of s to itself $n-1$ times. A special case is $s^0 = \lambda$. If Σ is an alphabet, then Σ^* is a set of all strings obtained by concatenating zero or more symbols from Σ. The set Σ^* contains

λ. The $\Sigma^+ = \Sigma^* - \{\lambda\}$ is defined for convenience. Σ^+ and Σ^+ are infinite. A *language* L is a subset of Σ^*. Operations for sets, such as union, intersection, difference and complements are valid for languages [5]. The complement of L is

$$\overline{L} = \Sigma^* - L. \tag{6.1.12}$$

The *reverse of a language* L is a set L^R of the reversed strings of L:

$$L^R = \{s^R : s \in L\}. \tag{6.1.13}$$

The *concatenation of languages* defined as follows

$$L_1 L_2 \cdots L_n = \{s_1 s_2 \cdots s_n : s_1 \in L_1, s_2 \in L_2, ..., s_n \in L_n\}. \tag{6.1.14}$$

A *grammar* G is defined as $G = \{V, T, S, P\}$, where

- V is a finite set of objects called *variables*,
- T is a finite set of objects called *terminal symbols*,
- $S \in V$ is the *start variable*,
- P is a finite set of *productions*.

If $G = \{V, T, S, P\}$ is a grammar, then

$$L(G) = \{s \in T^* : S \Rightarrow^{G.P^*} s\} \tag{6.1.15}$$

is the language generated by G. $S \Rightarrow^{G.P^*} s$ denotes that starting from the start variable S, we successively apply productions of G and arrive at the s sentence [5]. If $s \in L(G)$, then the sequence

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots s_n \Rightarrow s \tag{6.1.16}$$

is a *derivation* of the sentence s. The strings $S, s_1, s_2, ..., s_n$, which contain variables and terminals, are *sentential forms* of the derivation.

6.2 Deterministic Finite Accepters

A *deterministic finite accepter* is defined by the quintuple

$$A = (Q, \Sigma, \xi, q_0, F),$$

where

- Q is a finite set of *states*,
- Σ is a finite set of input symbols (alphabet),
- $\xi : Q \times \Sigma \to Q$ is the *transition function*,

- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

A deterministic finite acceptor operates (DFA) as follows: DFA starts at the initial state q_0 and is ready to accept an input symbol. When an input symbol from Σ arrives, transition function ξ determines the next state and the acceptor makes a transition to that state. When the whole input string is consumed and the acceptor is in one of the final states, the string is accepted, otherwise, it is rejected. The set of all strings on Σ accepted by $A = (Q, \Sigma, \xi, q_0, F)$ is the *language of acceptor* A:

$$L(A) = \{\omega \in \Sigma^* : \xi^*(q_0, \omega) \in F\},$$

where ξ^* is a recursive application of $\xi : \xi(\xi(q^0, \omega(0)), \omega(1)) \cdots$ [5].

6.3 Nondeterministic Finite Accepters

A *nondeterministic finite acceptor* (NFA) is defined by quintuple

$$A = (Q, \Sigma, \xi, q_0, F),$$

where

- Q is a finite set of *states*,
- Σ is a finite set of input symbols (alphabet),
- $\xi : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ is the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

There are a few differences between NFA and DFA. The transition function of NFA has range in 2^Q, so that its output may include more than one state in Q. This corresponds to nondeterministic transition to one of a few possible states. NFA can make a transition on an empty string λ. Range of the transition function ξ is 2^Q which includes an empty set, meaning that for some input symbols, next state may be undefined. For an input string to be accepted by NFA, there should be at least one possible path to one of the final states. The language of NFA defined as follows

$$L(A) = \{\omega \in \Sigma^* : \xi^*(q_0, \omega) \cap F \neq \emptyset\},$$

where ξ^* is a recursive application of ξ [5].

6.4 Equivalence of Deterministic and Nondeterministic Finite Acceptors

Two acceptors are said to be equivalent if they accept the same language: $L(A_1) = L(A_2)$. As any particular language can have many acceptors, an acceptor has many equivalent acceptors [5].

In [5], an algorithm is presented to build DFSA from corresponding NFSA. The idea behind the algorithm is that if NFSA performs nondeterministic state transition to one of the possible states, we create a state in DFSA which includes these states form NFSA. If a state in the DFSA created in this way includes one or more final states of the NFSA, we mark this state as final.

In figure 6.3 an example of NFSA is represented, which has the nondeterministic state transition from the state q_0.

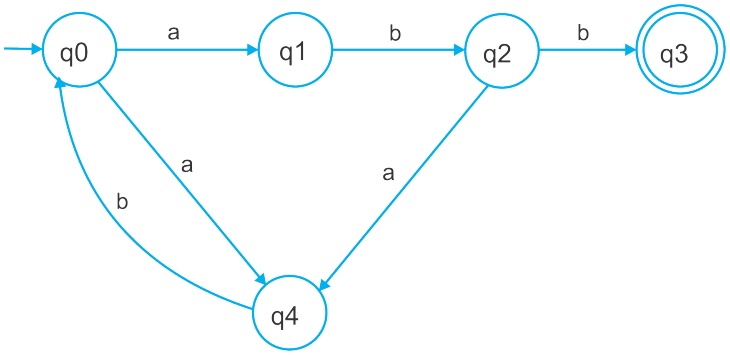


Figure 6.3: Example of NFSA.

The NFSA depicted in figure 6.3 can be described by so called *transition table* 6.2

Table 6.1: Transition table for the NFSA in figure 6.3.

Current state/Input	a	b
q_0	$\{q_1, q_4\}$	{ }
q_1	{ }	$\{q_2\}$
q_2	$\{q_4\}$	$\{q_3\}$
q_3 (Accept state)	{ } or we can define that after a string has been accepted the NFSM goes to the initial state and is ready to accept new strings	{ }
q_4	{ }	$\{q_0\}$

In figure 6.4 a DFSA is represented, which corresponds to the NFSA depicted in figure 6.3.

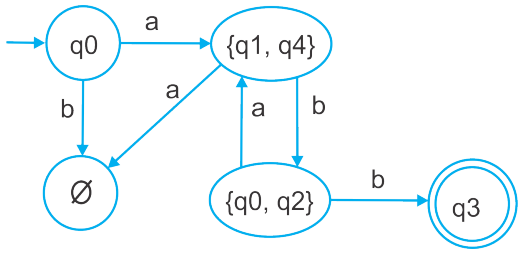


Figure 6.4: DFSA built form NFSA.

The construction of the DFSA depicted in figure 6.4 is performed as follows:

1. Initial state q_0 from the NFSA is added to the DFSA as initial state q_0.

2. From state q_0 in the NFSA on input "a" there is nondeterministic state transition to the states q_1 and q_4. To represent that in the DFSA we add state $\{q_1, q_4\}$, which is reached from state q_0 when input "a" arrives (figure 6.4).

3. Transition from q_0 of NFSA on input "b" is not defined. We represent that with the state $\emptyset$ in the DFSA. Which, in essence, acts as a trap state.

4. Now we consider our newly created state $\{q_1, q_4\}$. This state represents that the corresponding NFSA can be in q_1 or q_4. Form q_1 the NFSA can make a transition to q_2 on input "b", and from q_4 to q_0 on input "b". We create a new state $\{q_0, q_2\}$ in the DFSA to represent this. The transition is performed from the state $\{q_1, q_4\}$ to the state $\{q_0, q_2\}$ on input "b". From the state $\{q_1, q_4\}$ there is no transition defined on input "a", so we add a transition from it to the sate $\emptyset$.

5. At the last step, we created the state $\{q_0, q_2\}$. Let us determine where from this state the DFSA can make a transition. We repeat what we did in the previous step. First, we consider the state q_0. It has nondeterministic transition on input "a" to the states q_1 and q_4, transition on input "b" is undefined. Then we consider the state q_2. It has transition to the state q_4 on input "a", and transition to the sate q_3 on input "b". To reflect that in our DFSA, we add transition from the state $\{q_0, q_2\}$ to the state $\{q_1, q_4\}$ on input "a", and transition to the final state q_3 on input "b".

6. The construction is complete, since for all created sates transitions are defined.

The NFSA and the DFSA, depicted in figures 6.3 and 6.4 respectively, are equivalent.

6.5 Comparison between NFSM and DFSM

In the previous sections, we formally defined and discussed NFSA and DFSA. Their main purpose is to represent regular expressions by performing regular expression matching [6]. For the purposes of this work, it is needed to extend NFSA with outputs on state transitions. The extended version of NFSA we will call Nondeterministic Finite State Machine NFSM. It is defined by

$$M = (Q, \Sigma, \xi, q_0, F, \Psi, \gamma), \tag{6.5.1}$$

where

- Q is a finite set of *states*,
- Σ is a finite set of input symbols (alphabet),
- $\xi : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ is the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*,
- Ψ is a finite set of output symbols,
- $\gamma : Q \times \Sigma \to \Psi$ is the output function.

This is a Mealy machine, since the output depends on current state and input, as opposed to a Moore machine, where output depends only on current state.
In section "Equivalence of Deterministic and Nondeterministic Finite Acceptors", it is stated that nondeterministic and deterministic finite state acceptors are equivalent (which generate no outputs). But when we introduce outputs on state transitions, the equivalence is no longer valid. It can easily be seen from the figures 6.3 and 6.4. For example, if we apply input "ab" to the NFSM, it will be either in state q_0 or q_2. If then we apply input "b", the NFSM will be either in the final state q_3 or in the trap state $\emptyset$. The output is generated nondeterministically, but it is know what possibilities be have. Now lets consider the corresponding DFSM. After we apply input "ab", the DFSM is in state $q_0, 1_2$, and after we apply input "b", it goes to the final state. There is no possible transition to $\emptyset$, the information is lost.
Further on, we will work only with NFSM.

6.6 Execution of NFSM

Now that we have defined NFSM by 6.5.1, we can consider its execution.
Let us consider the NFSM depicted in figure 6.5.
As can be seen from figure 6.5, there are outputs defined on each transition. Fo example, if the NFSM is in the initial state q_0 and input "a" occurs, the possible outputs are "x" and "z". The output alphabet is $\{x, y, z\}$, and the input alphabet is $\{a, b\}$. For an NFSM to accept a string, after reading the

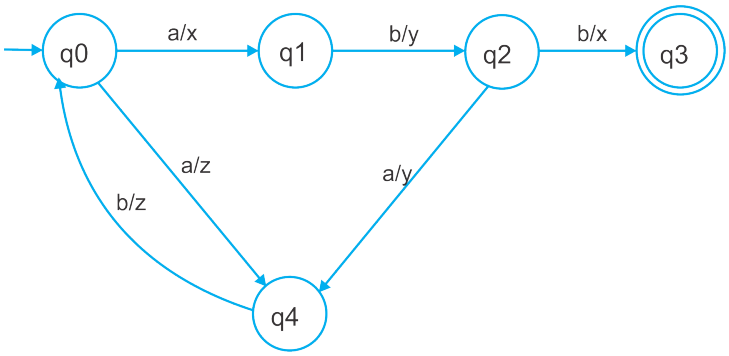


Figure 6.5: Example of NFSM.

string there should be at least one path leading to the accept state. When executing an NFSM in a software or in hardware, each nondeterministic state transition produces one or more copies of the NFSM, which run in parallel. A string is accepted when one of the NFSM reaches accept state.
As an example, we execute the NFSM depicted in figure 6.5. Initially there is just of copy of NFSM in the current sate q_0 represented by the filled circle in figure 6.6. Then input "a" is applied, and one copy is created with the current state $q1$. The initial NFSM is in the state q_4.

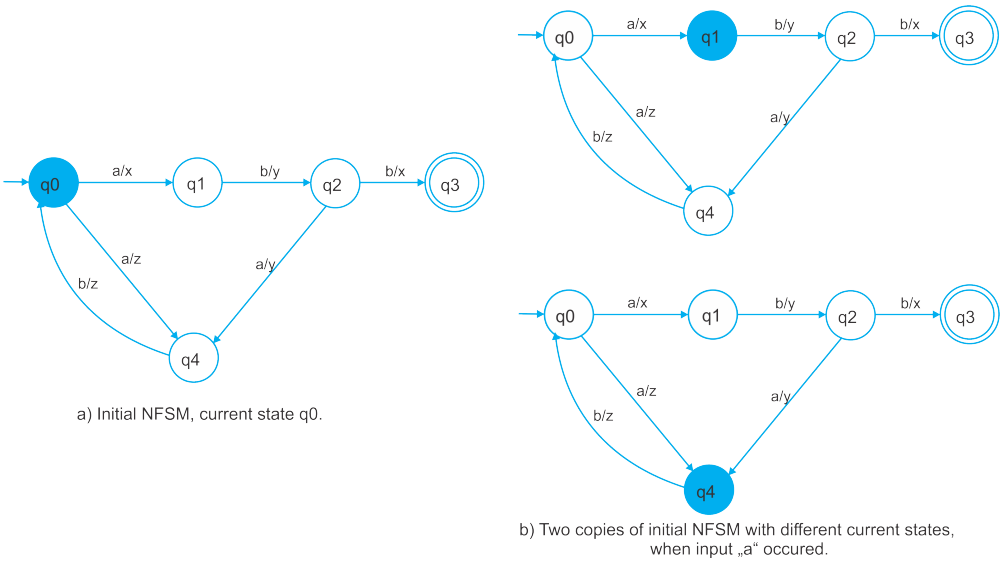


Figure 6.6: Execution of NFSM, input "a".

Continuing the execution, after input "b" arrives the two copies of the NFSM are depicted in figure 6.7.

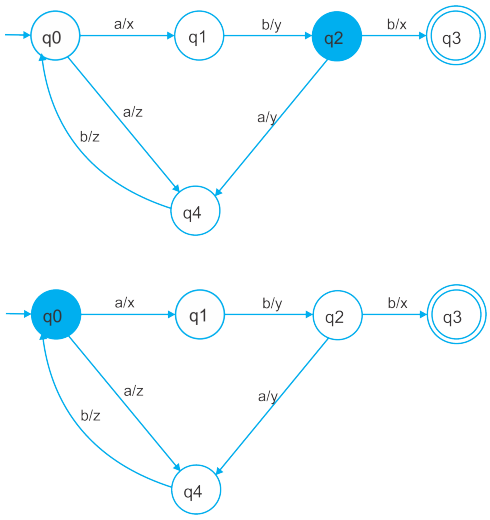


Figure 6.7: Execution of NFSM, input "ab".

As can be seen in figure 6.7, one copy of the NFSM is in the state q_3, and if input "b" occurs, it will accept the string "abb". The other copy of the NFSM is in the initial state q_0, and on input "a" will copy itself the same way as depicted in figure 6.6. From that follows that a string "abababab..." will produce $(number\ of\ "a")^2$ copies of NFSM. This observation does not apply, of course, to an arbitrary string.
One of the main objectives of this thesis is to develop an algorithm which computes maximum number of NFSM instances, given regular expression and limit on input string length.

6.7 Regular Expressions

In this section, regular expression syntax is introduced. We use regular expressions to describe control logic. For this purpose we need to extend regular expressions with actions (output in NFSM). But first, we consider regular expressions without actions. A regular expression represents a pattern of strings. The set of strings that fit the pattern defined by a regular expression form a regular language. Throughout this thesis, we will work with some features of POSIX basic regular expressions (BRE) and extended regular expressions (ERE). According to [7], [8], and [9], POSIX BRE and ERE syntax is defined as follows. POSIX BRE and ERE is composed of characters (a-z, A-Z, 0-9, etc.) and/or meta-characters. Characters in a regular expression match the same characters in a string. For example, the regular expression "a" matches the string {a}; regular expression "abcd" matches the string {abcd}. In the table 6.2, POSIX BRE and ERE meta-characters are listed, which are used in this work.

Table 6.2: POSIX BRE and ERE meta-characters

Meta-character	Description	Example
"."	Matches any single character.	For example, "a.b" matches {aab, abb, acb, ... }.
"|"	Represents a choice between two patterns. Logical OR.	"ab|cd" matches either {ab, cd}.
"()"	Defines a subexpression.	"ab(c|d)ef" matches either {abcef} or {abdef}.
"*"	Matches the preceding character or subexpression zero or more times.	"abc*d" matches {abd, abcd, abccd, abcccd, ...}.
"+"	Matches the preceding character or subexpression one or more times.	"abc+d" matches {abcd, abccd, abcccd, ...}.
"?"	Matches the preceding character or subexpression zero or one times.	"abc+d" matches {abd, abcd}.

Now we will extend the regular expression syntax with actions.

6.8 Thompson's algorithm: From regular expression to NFSM.

Ken Thompson in his paper [10] proposed an algorithm, which converts a given regular expression to the corresponding NFSM. In [10], the Thompson's algorithm is adapted for implementation in a compiler. The adaptation (transformation to reverse Polish form, etc.) is not relevant for the purpose of this work, so it is presented here in a simplified version. We will present the same example used in [10]. Let us convert the regular expression $a(b|c)^*d$ to the corresponding NFSM using a simplified version of Thompson's algorithm. The algorithm iteratively constructs the NFSM from a regular expression. First, the NFSMs matching each single non-metacharacter of the regular expression are constructed (figure 6.8) [11].

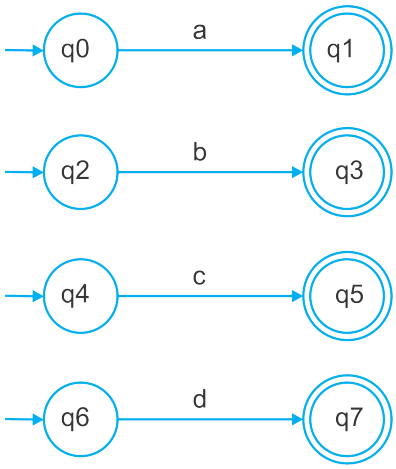


Figure 6.8: Thompson's algorithm: first iteration.

In the second iteration, the algorithm goes up the regular expression hierarchy and constructs the NFSM for $b|c$ using the NFSMs for b and c constructed in the previous iteration. The NFSM for $a|b$ is depicted in figure 6.9. Note, that there is just one final state. Here λ transitions have been used. As is described in the section "Formal languages", $\lambda - transitions$ consume no input symbol. Now the question may arise as to how to implement $\lambda - transitions$. The answer to this is simple: $\lambda - transitions$ are always executed. If there is $\lambda - transitions$ and another ordinary transition coming from the same state, it will be considered a nondeterministic transition if the ordinary transition is performed. Nondeterministic transitions are executed as described in the section "Execution of NFSM".

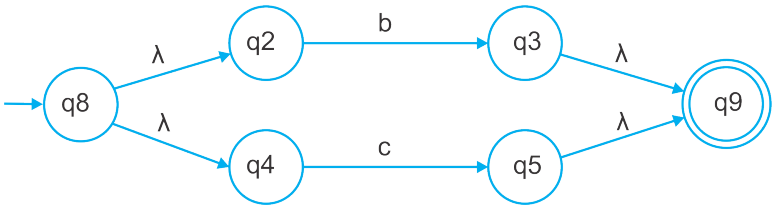
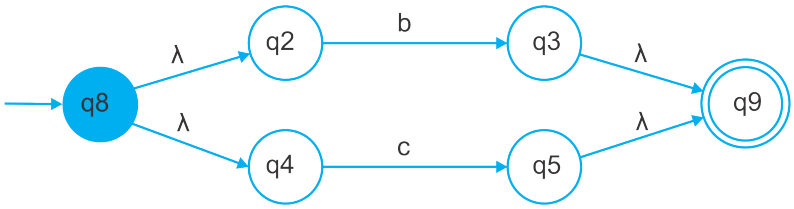
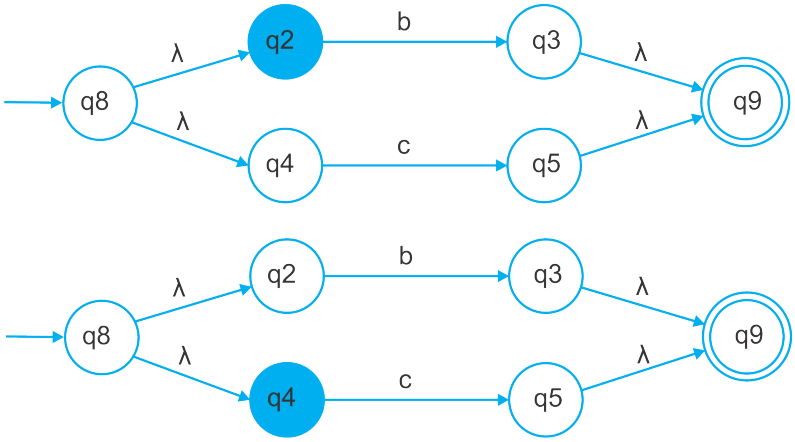


Figure 6.9: Thompson's algorithm: second iteration.

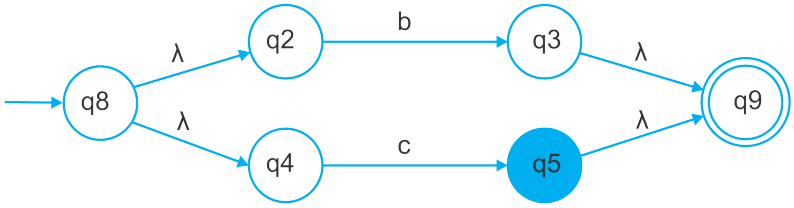
Lets us, for example, execute the NFSM depicted in figure 6.9. The execution is depicted in figure 6.10.



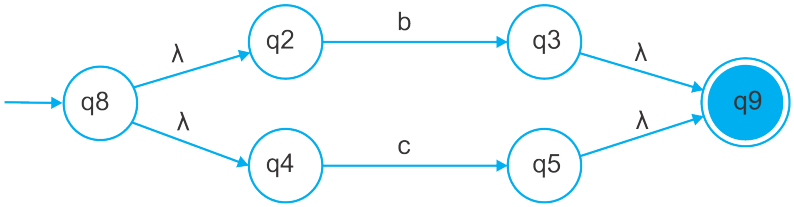
a) Starting in the initial state.



b) Nondeterministic λ-transitions performed, no input symbol has been consumed. Two copies of the NFSM running in parallel.



c) Input symbol „c“ has been consumed. Only one copy of the NFSM is left, since from the state q2 there is no transition defined for the symbol „c“ and this copy of the NFSM is removed.



d) λ-transition is performed (they are always executed) and the accept state is reached. The string „c“ is accepted. It is not hard to see that the string „b“ would be also accepted. So the NFSM matches the regular expression „c|b“.

Figure 6.10: Execution of the NFSM matching the regular expression $c|b$.

In the third recursion, the NFSM for $(b|c)^*$ is constructed (figure 6.11).

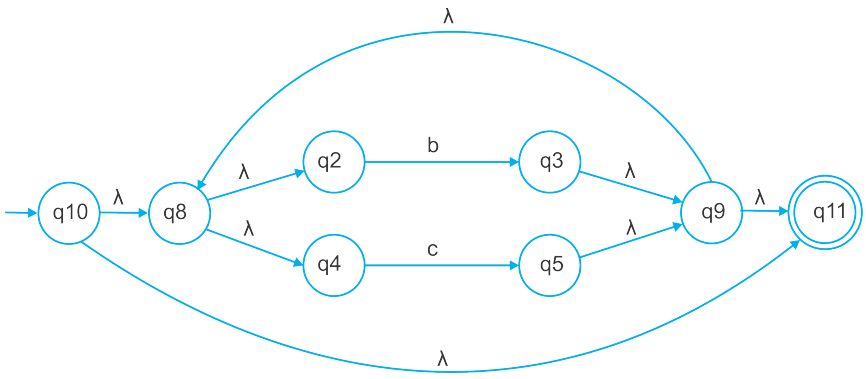
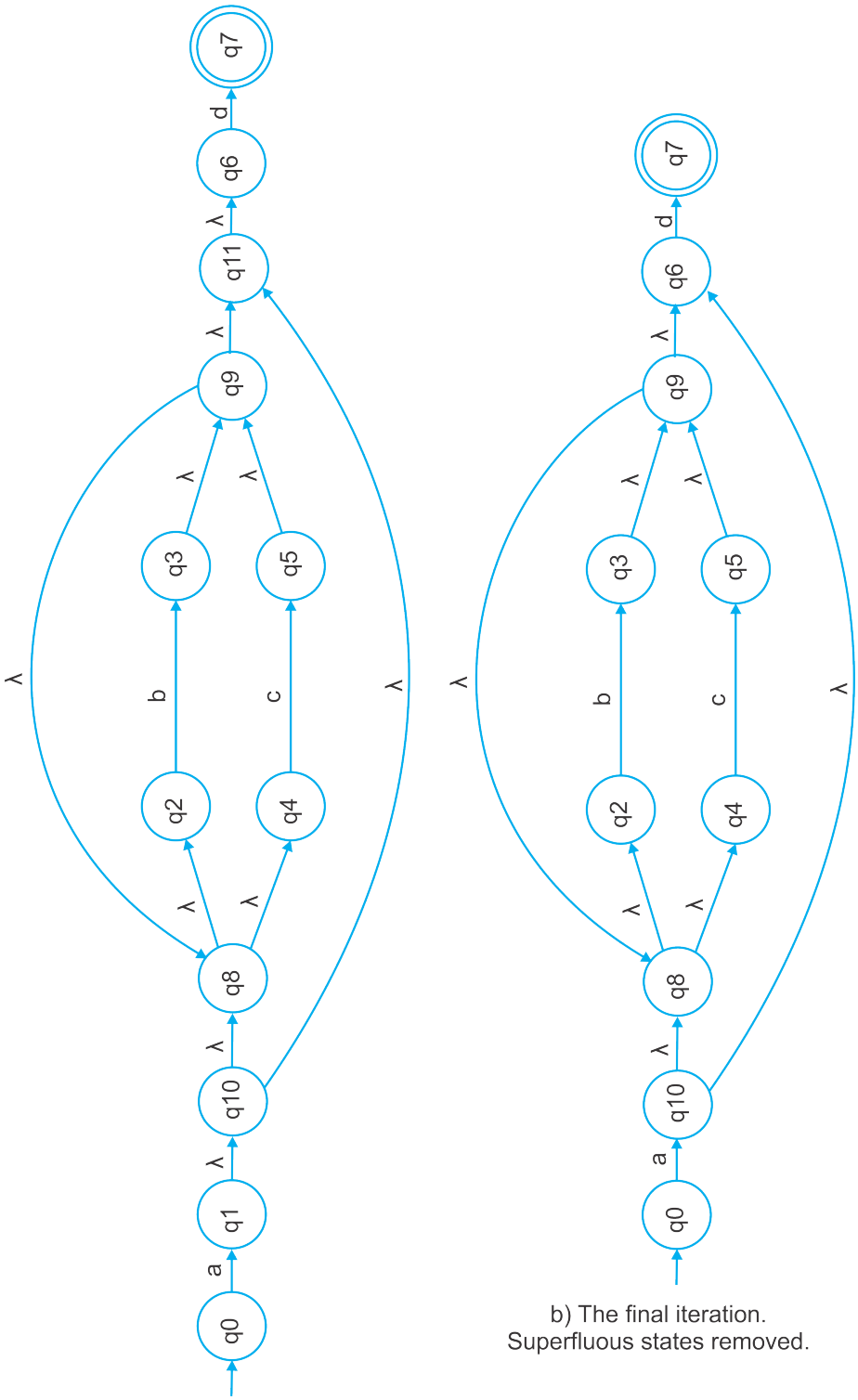


Figure 6.11: Thompson's algorithm: third iteration.

In the figure 6.11, the λ-transition from q10 to q11 corresponds to zero of $(b|c)$ matched as defined for the "*" metacharacter (see table 6.2). The λ-transition from q9 to q8 corresponds to $(b|c)$ matched many times.
In the fourth iteration, the algorithm concatenates the constructed NFSMs in previous iterations together producing the final NFSM depicted in figure 6.12.



a) The final iteration. As can be seen, q1 and q11 are superfluous and can be removed.

b) The final iteration. Superfluous states removed.

Figure 6.12: Thompson's algorithm: fourth iteration, final.

Above we constructed NFSMs for |, (), and $*$ meta-characters. Now we will construct NFSMs for the rest of meta-characters. In figure 6.13 the NFSM for the $a+$ regular expression is depicted.

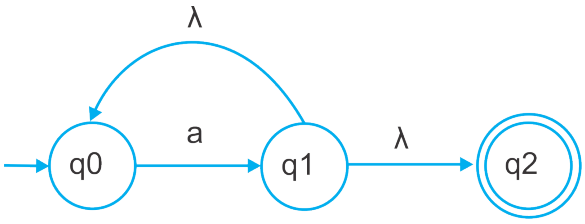


Figure 6.13: Thompson's algorithm: NFSM for "a+".

The NFSM for the "a?" regular expression is depicted in figure 6.14.

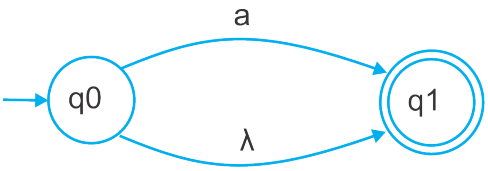


Figure 6.14: Thompson's algorithm: NFSM for "a?".

Let us now construct the NFSM for the "a." regular expression (figure 6.15). In this case, we cannot use λ-transition, since it does not consume input symbols, and we need to consume one. The solution is simple, but not elegant: we create transitions for each symbol in the input alphabet from the state reached after reading symbol "a" to the final state. The more elegant and easy implementable in software solution is to use the λ-transition, which consumes one symbol. Let us identify it as β-transition. The NFSM for the "a." regular expression constructed using the β-transition is depicted in figure 6.16.

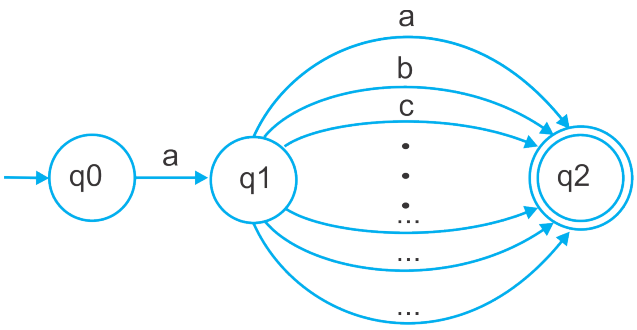


Figure 6.15: Thompson's algorithm: NFSM for "a.".

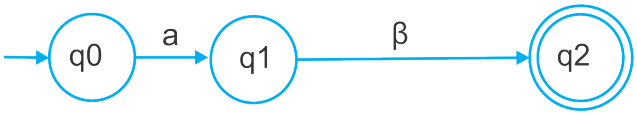


Figure 6.16: Thompson's algorithm: NFSM for "a.", using β-transition.

Note, that the resulting FSM is deterministic.

Chapter 7

Computing bounds for NFSM execution

In the previous chapter, we demonstrated some examples of execution of an NFSM. In figure 6.3 it can be seen that when the infinite input string "abababababab..." is applied, there is infinite number of NFSM copies, considering that on each nondeterministic state transition a new copy is created. In this chapter, we will develop an algorithm which computes maximum number of NFSM copies that can be created during its execution, given a regular expression. This is very important for the other master project where we implement an NFSM on an FPGA, since an FPGA has limited number of hardware resources. The fact, that during execution some copies are deleted (undefined transition for the current input symbol) makes the work more complicated. There are two common approaches to this task. First, we can use simulation to determine the maximum number of copies. Simulation can give the most accurate results, but requires substantial amount of time to perform. Second, we can use some formal technique. The approach based on simulation is pretty straight-forward. For the purpose of computations of bounds, a computer program has been developed as a part of this work. The program accepts a regular expression, performs checks on correctness of the regular expression, and transforms the regular expression into the corresponding NFSM. After transformation is complete, the program can perform the following actions:

- Perform matching on a single string entered by user.
- Simulate NFSM with random strings of specific length, giving as a result maximum number of NFSM copies.
- Apply formal method (algorithm) to compute maximum number of NFSM copies.

The user interface of the program is depicted in figure 7.1.

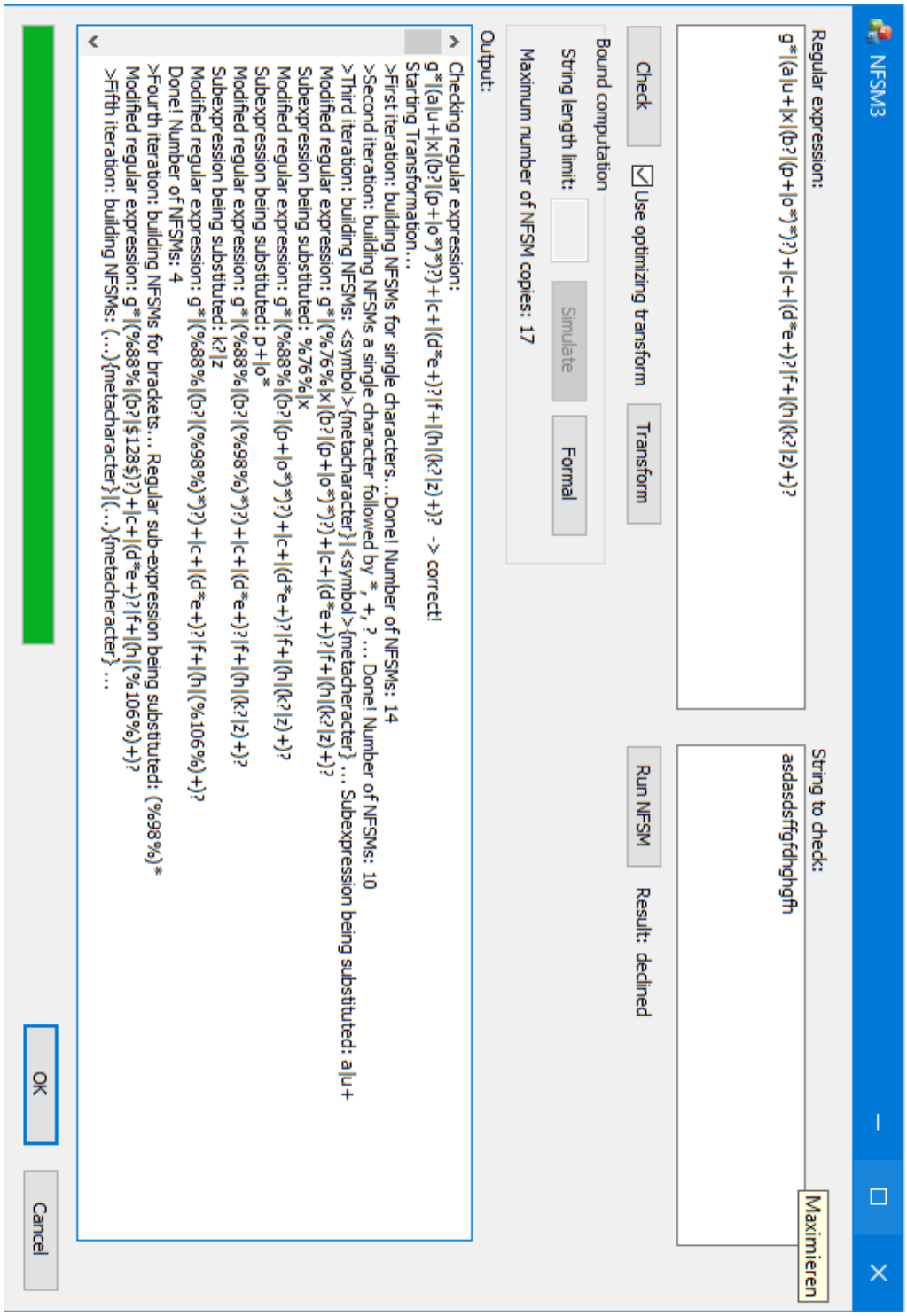


Figure 7.1: User interface.

In figure 7.1, the text-field named "Regular expression" is used to enter a regular expression for conversion. After a regular expression is entered, the button "Check" is pressed, which triggers checking procedure. The procedure performs syntactical analysis of the regular expression. If the regular expression is syntactically correct, the button "Transform" becomes active, and when pressed, runs the modified Thompson's transformation algorithm on the regular expression. As a result the program contains an NFSM for the regular expression. The NFSM is stored in the memory of the program as a set of states connected by pointers. At this stage, the program does not have an option to save NFSMs between runs, though it has ability to save the graph of the NFSM in a "DOT" format. After the regular expression has been transformed to the corresponding NFSM, we can perform matching of strings with it. In the program (figure 7.1), there is the text-field named "String to check", which accepts one string upon which matching is performed. After the button "Run NFSM" is pressed we should have the result of matching displayed after the text "Result:"; It is either "accepted" or "declined". While performing checking, transformation, and matching the program displays output in the text-field "Output". The most important: the output contains information about the transformation process on different stages:

- The first stage. NFSMs for each non-metasymbol in a regular expression are constructed. After completion, the number of NFSMs constructed that way is displayed, it should be equal to the number of distinct non-metasymbols in the regular expression.
- The second stage. NFSMs for symbols followed by a metacharacter (*, +, ?) are constructed. As with the first stage, at the end the number of NFSMs is displayed. This stage uses the NFSMs constructed in the first stage.
- Third stage. NFSMs for $symbol\{metacharacter\}|symbol\{metacharacter\}$ are built. After NFSM for one of the "—" regular expressions is built, the regular expression is substituted with "%$number$%". This can be seen in "Output" text-field in figure 7.1. For example, the regular expression "a*|b" is substituted with "%19%".
- Fourth stage. NFSMs for pairs of brackets followed by a metacharacter are built. The program is able to handle complex hierarchies of brackets. After NFSM for a pair of brackets is built, the brackets with the content inside is substituted with "$number$". For example, the regular expression "(abc*d)+" is substituted with "21".
- Fifth stage. NFSMs for "$(\dots)metacharacter|(\dots)metacharacter$" are built. The regular expressions are substituted with "%number%".
- Sixth stage. The final NFSM is built by concatenation of the previously built NFSMs.

Note: the stages can be repeated in a different order a few times during the transformation. The need for this behavior is following. Consider regular expression $(a(b|c)+|(m|n)+k)+$. The first stage creates NFSMs for single symbols a, b, c, m, n, and k. The second stage does nothing as there are no symbols followed by metacharacter. The third stage creates NFSMs for "b—c" and "m—n" and substitutes them in original regular expression with "%24%" and "%30%", correspondingly (figure 7.1, "Output"-field). This creates the regular expression $(a(\%24\%)+|(\%30\%)+k)+$. On this regular expression works the fourth stage, which creates NFSMs for pairs of brackets followed by a metacharacter. But the fourth stage can create NFSMs only for $(\%24\%)+$ and $(\%30\%)+$, since the outermost pair of brackets cannot be constructed at this moment. The fifth stage constructs NFSM for $(\%24\%)+|(\%30\%)+$ and now the fourth stage resumes its work and constructs NFSM for $(a(\%24\%)+|(\%30\%)+k)+$. The concatenation (the sixth stage does nothing in this case).
When the transformation is complete, it is possible to check the correctness of the transformation by examining the output. Since the most problematic part is hierarchies of brackets combined with "|", it is worth while always to check that substitution is performed in correct order.

7.1 Construction of graph diagram.

A graphical representation of an NFSM is very useful for checking correctness of transformation and for other analytical purposes. In this section, we describe the process of visualizing an NFSM and the tools used.
For drawing an NFSM diagram, we use free open source software GraphViz, which accepts "DOT" format [12]. After the program, developed in this work, performed transformation of a regular expression to the corresponding NFSM, it creates a file named "output.txt" in the directory of the program, where it writes the NFSM structure using "DOT" format. "DOT" format is very simple and captures just structural information of an NFSM, It contains no information about actual geometrical properties of the diagram.
After the "output.txt" file has been generated, we use GraphViz program, which reads this file, to draw the diagram.
The window of the GraphViz program is depicted in the figure 7.3. GraphViz reads a file in "DOT" format, performs necessary optimizations, and draws the diagram. It is possible to save the diagram in many file formats, among them .pdf, .svg, .png.
For the diagram depicted in the figure 7.4, the DOT-file has the contents depicted in figure 7.2.
The documentation and the user guide for GraphViz can be found on the website [12].

```
strict digraph {
        rankdir = LR
        122 -> 124 [label=§]
        124 -> 125 [label=a]
        125 -> 126 [label=§]
        126 -> 127 [label=§]
        126 -> 128 [label=§]
        127 -> 129 [label=§]
        128 -> 130 [label=§]
        129 -> 131 [label=§]
        129 -> 132 [label=§]
        130 -> 133 [label=§]
        130 -> 134 [label=§]
        131 -> 135 [label=b]
        132 -> 136 [label=c]
        133 -> 137 [label=m]
        134 -> 138 [label=n]
        135 -> 139 [label=§]
        136 -> 139 [label=§]
        137 -> 140 [label=§]
        138 -> 140 [label=§]
        139 -> 141 [label=§]
        139 -> 129 [label=§]
        140 -> 142 [label=§]
        140 -> 130 [label=§]
        141 -> 143 [label=§]
        142 -> 143 [label=§]
        143 -> 144 [label=§]
        144 -> 145 [label=k]
        145 -> 123. [label=§]
        145 -> 124 [label=§]
}
```

Figure 7.2: DOT-file contents.

In the diagram depicted in figure 7.4, it can be seen that there are some superfluous states, for example, 126 , 127, 128, etc. These states do not change the NFSM, they are introduced by the transformation algorithm.

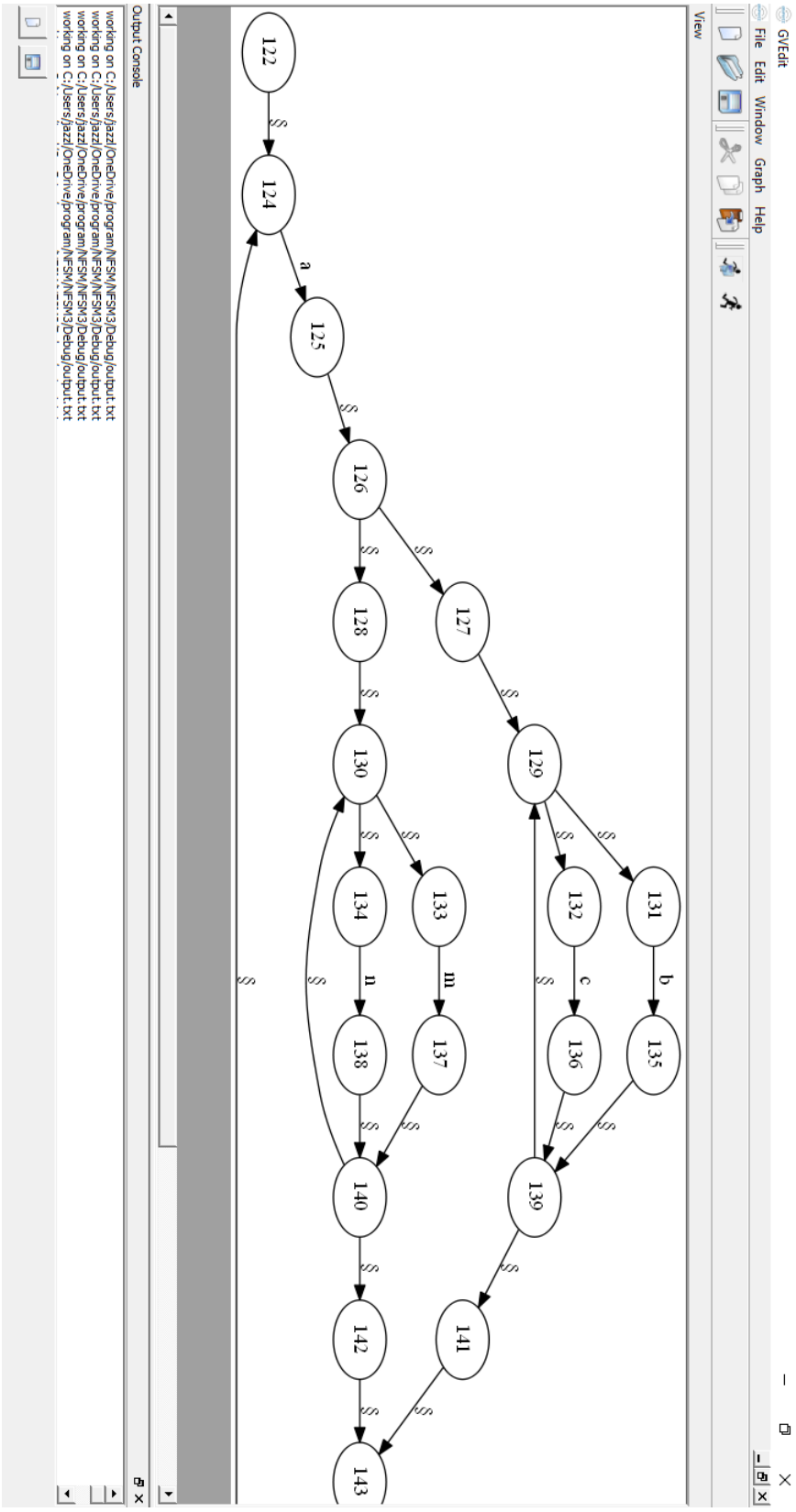


Figure 7.3: GraphViz window.

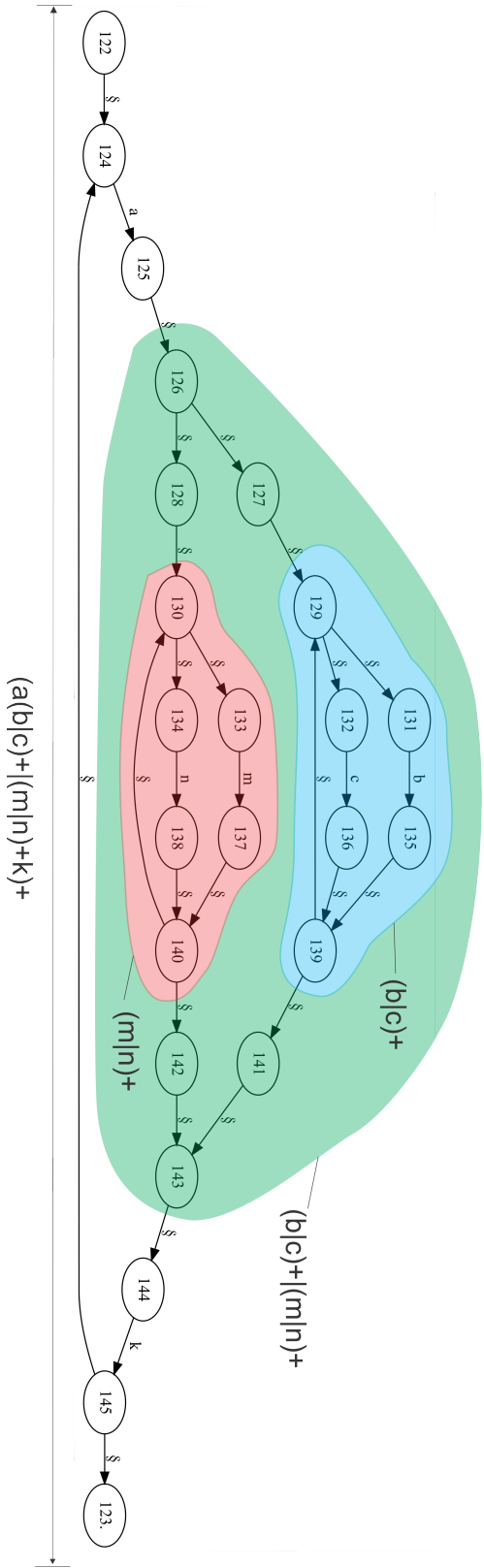


Figure 7.4: Graph constructed by the program. § represents lambda-transition, "." after the number of the state represents the final state.

There is an option in the program to perform optimizing transformation, which removes superfluous states from the final NFSM. To enable the optimizing transformation, simply check the checkbox "use optimizing transform". The NFSM diagram for the regular expression $(a(b|c)+|(m|n)+k)+$ constructed using the optimizing transformation is depicted in figure 7.5. As can be seen in figure 7.5, there are no superfluous states.

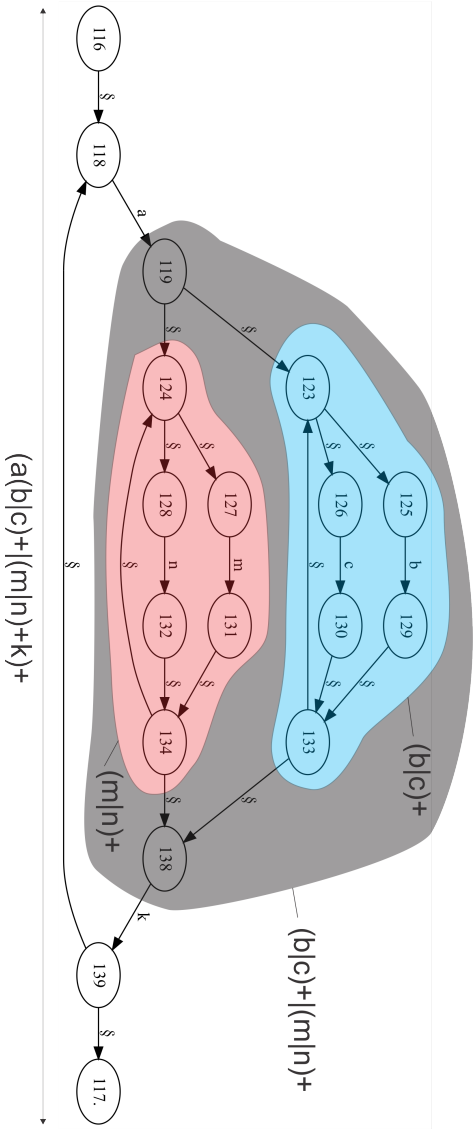


Figure 7.5: Graph constructed by the program (optimization is on). § represents lambda-transition, "." after the number of the state represents the final state.

In the following subsections, methods for computing maximum number (upper bound) of copies of an NFSM in its execution are described.

7.2 Computing bounds using simulation

Computing bounds using simulation is straight-forward. Given a regular expression and a limit on string length, on which matching is performed, it is required to determine the maximum number of copies of the NFSM when it is run on the string. As is mentioned before, an NFSM is copied whenever a non-deterministic state transition is performed. Moreover, a copy of an NFSM is deleted either when there is no transition specified from the current state for an input symbol read, or the transition leads to the final state, but there are symbols still left to be read. A string is accepted only when the last symbol read leads to the final state. This principle is implemented in the string matching mechanism in the program developed in this work.
To determine the maximum number of copies of an NFSM by simulation, the following steps are performed:

1. Generate a random string of specified length, which differs from previously generated strings.
2. Simulate an NFSM with the random string generated in the first step.
3. Record maximum number of copies of the NFSM produced during simulation in the second step.
4. If the maximum number of copies received in the third step is higher than the number from the previous simulation, update the maximum, if not, store the previous value.
5. Repeat from step one, until all combinations of strings of specified length are simulated.

It is easy to see, that the simulation method is very slow, since the number of strings to simulate grows exponentially with the string length. To reduce the simulation time, some heuristics can be used. In this case heuristics will choose some strings that produce the most copies of an NFSM, so the number of strings to simulate will be reduced.
In this work, the method based on simulation is not implemented in the program, because simulation takes long time to complete.

7.3 Formal method

By formal method, we mean a method which is not a brute-force one as simulation and relies on structure of an NFSM diagram, which is directed graph, to determine maximum number of an NFSM copies in execution.
NFSM diagrams can be divided into two categories:

- Diagrams built using the Thompson's algorithm.
- Diagrams which cannot be built by the Thompson's algorithm.

In this section, we will create an algorithm for computing maximum number of NFSM copies for an NFSM diagrams built using the Thompson's algorithm.
The NFSM diagram depicted in figure 6.3 is an example of an NFSM diagram which cannot be built using the Thompson's algorithm. Devising a formal method for computing bounds on number of copies for this kind of diagrams is much more involved than for the other kind.
The Thompson's algorithm works by connecting in parallel or in series limited number of "standard" NFSMs depicted in figures 6.13, 6.14, 6.15, etc. This leads to the fact that NFSMs constructed by the Thompson's algorithm have regular structure and are simple for analysis. Let us further on refer to this kind of NFSM diagrams "regular NFSM".
Let us examine a regular NFSM diagram depicted in figure 7.6

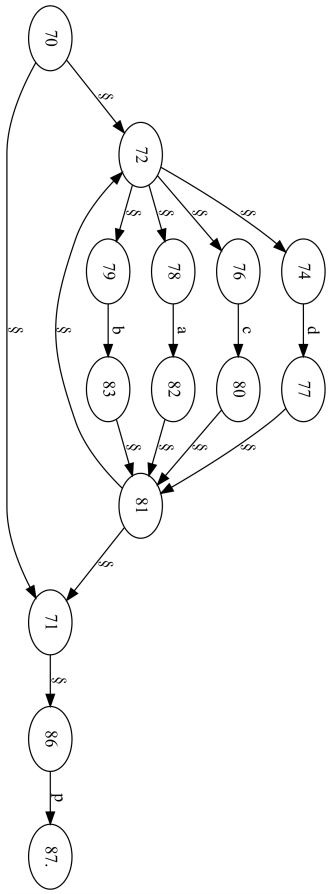


Figure 7.6: Regular NFSM diagram. § represents lambda-transition, "." after the number of the state represents the final state.

In the figure 7.6, the NFSM is constructed from the regular expression $(a|b|c|d)*p$. If we try to execute this NFSM, we get a bound on copies no more than five, no matter what string length is. This is because of the regular structure of the NFSM and execution rules, which for convenience presented bellow. Considering the following execution rules:

- an NFSM is copied on nondeterministic state transitions.
- an NFSM copy is deleted when there is no specified transition for a current state and a symbol read.
- an NFSM copy is deleted when the final state reached, but there are still symbols to read.
- If there exists a §-cycle, the last §-transition returning to the initial state of the cycle is not performed.

- String is accepted when after reading all its characters one of the copies of an NFSM reaches the final state.

We can analyze the NFSM depicted in figure 7.6 and determine that the bound is indeed equal five. Starting from the initial state, we perform §-transitions (also known as lambda-moves or epsilon-moves) and reach the following sates: 74, 76, 78, 79, and 86. So we have five copies of the NFSM, each of which is in different current state. Then one symbol is read and two possibilities exist:

- There is a transition defined for this symbol from the one of the current states.
- There is no such transition.

So if the symbol read is "a", then only one NFSM copy is left, which performs transition and is in current state 82 (figure 7.6). Other NFSM copies are deleted. If for example, symbol "k" is read, then all NFSM copies are deleted and the string is not accepted. As for the "loop"-transition from the state 81 to the state 72, it does not present a problem. If we perform §-transitions from the current state 82, we reach the same states 74, 76, 78, 79, and 86, which corresponds to five copies of the NFSM. As there are no other possibilities to increase the number of copies, we proved that the bound for the NFSM is five copies. The idea here is that: since cyclic subgraphs in an NFSM diagram are the only source of potential NFSM copies growth, and since in regular NFSMs they, cyclic subgraphs, have regular structure, which does not allow copies growth, there exist a bound on number of copies for any string length, if execution is performed according to the rules.
Note: the above statement is true only if an NFSM does not have cyclic OR-subgrapths with two or more transitions with the same symbol (figure 7.7).
If NFSM constructed by the Thompson's algorithm has one or more cyclic OR-subgrapths with two or more transitions labeled with the same symbol, we name such NFSM pseudo-regular.

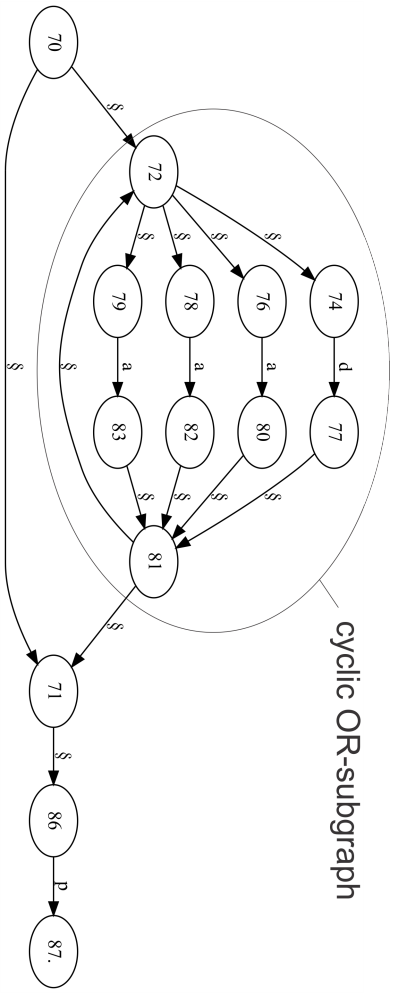


Figure 7.7: Pseudo-regular NFSM diagram. § represents lambda-transition, "." after the number of the state represents the final state.

In the case of NFSM depicted in figure 7.7, when "a" symbol is read, three NFSM copies are left. These three copies then each make §-transitions and reach 74, 76, 78, 79, and 86 states, totaling in fifteen copies. If we match the string "aaaaap" with this NFSM, we get 405 copies. In this case bound is dependent on string length. This kind of NFSMs are not considered in this work.

Now lets us consider the regular NFSM diagram depicted in figure 7.8. The diagram contains a §-loop between 86 and 95 states. According to the execution rules if the §-transition from the state 86 to the state 95 is performed, the §-transition from the state 95 to the state 86 is not performed for the same copy of NFSM. But transitions from the state 86 through the state 95 to the state

85 and transitions from the state 91 through the state 95 to the state 86 are allowed by the execution rules. This rule avoids execution of infinite cycles.

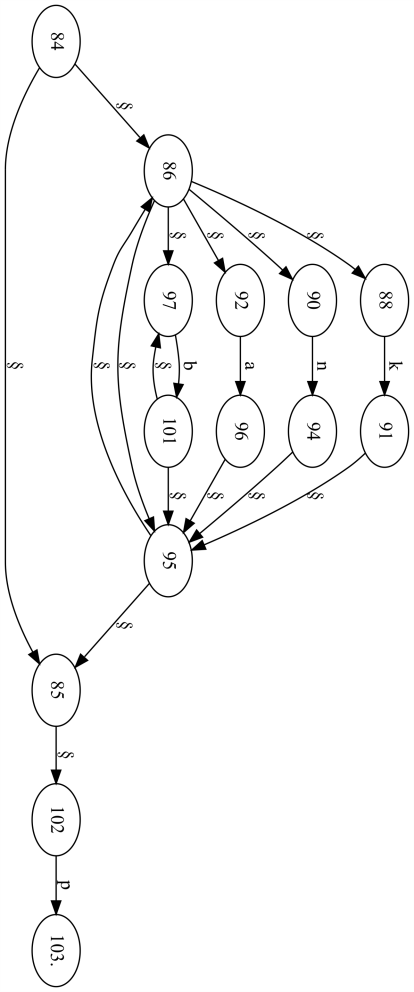


Figure 7.8: Regular NFSM diagram, example. § represents lambda-transition, "." after the number of the state represents the final state.

From all the observations above, we can start building the bound computation algorithm. The maximum number of NFSM copies equals to the maximum number of reachable states by §-transitions from some state of the NFSM. In the NFSM depicted in figure 7.8, the maximum number of reachable states by §-transitions is six, and it is found at the first evaluation (starting at the initial state 84). In more complex NFSMs, the maximum number of reachable states by §-transitions may not be found at the first evaluation. From this follows that the algorithm should traverse NFSM from initial to the final state to find the

absolute maximum.
The algorithm has the following steps:

1. Starting from an initial state, perform all possible §-transitions.
2. Record the number of reached states in the step 1. At this point, all the reached states have only non-§-transitions.
3. Create a set of symbols, which contain the symbols of the transitions of the states reached in the step 1.
4. For each symbol of the set, make a copy of the NFSM with the current states (one NFSM can have more than one current state here) that have been reached in the step 1, and feed a symbol to the NFSM. After the symbol is read, the NFSM performs transitions for the symbol and reaches new current states, those current states that have no specified transitions for the symbol read are marked as not current. NFSM is deleted if one of the transition leads to a final state or it looses the last current state (marked as non-current in the step).
5. Repeat from the step 1 for each NFSM until there is no NFSMs left.

Lets us perform this algorithm on the NFSM depicted in the figure 7.8.
First step, performing all §-transitions from the initial state 84, the following states are reached: 88, 90, 92, 97, and 102. But since the state 102 is reached by two different paths, we count it two times, and the number of reached states is six, as of this iteration.
At the second step, the number of reached states at the first step is saved, which is six.
Third step, the following set of symbols is created $\{a, b, n, k, p\}$. Note: symbols cannot be repeated in the set.
Fourth step:

1. Copy of the NFSM is created with current states 88, 90, 92, 97, and 102, and symbol "a" is read. After the symbol "a" is read, the NFSM has only one current state 96.
2. Copy of the NFSM is created with current states 88, 90, 92, 97, and 102, and symbol "b" is read. After the symbol "b" is read, the NFSM has only one current state 101.
3. Copy of the NFSM is created with current states 88, 90, 92, 97, and 102, and symbol "n" is read. After the symbol "n" is read, the NFSM has only one current state 94.
4. Copy of the NFSM is created with current states 88, 90, 92, 97, and 102, and symbol "k" is read. After the symbol "k" is read, the NFSM has only one current state 91.

5. Copy of the NFSM is created with current states 88, 90, 92, 97, and 102, and symbol "p" is read. After the symbol "p" is read, the NFSM has only one current state 103. This NFSM has reached the final state and is deleted.

At this point, the algorithm makes new iteration starting from the first step. The algorithm iterates until all possible paths are executed. This is insured by the step four. The stoping condition is that all NFSMs either reach the final state or make a cycle transition.
The formal method is implemented in the program. It has been tested with different NFSM complexities and proved to be accurate. Bellow a few examples are presented, where the formal method is used. The example NFSMs are chosen to present implementation-specific regular expression syntax.
First example is for $a|(b|v|g|nm|h*)|$ c regular expression. The NFSM diagram for this regular expression is depicted in figure 7.9.

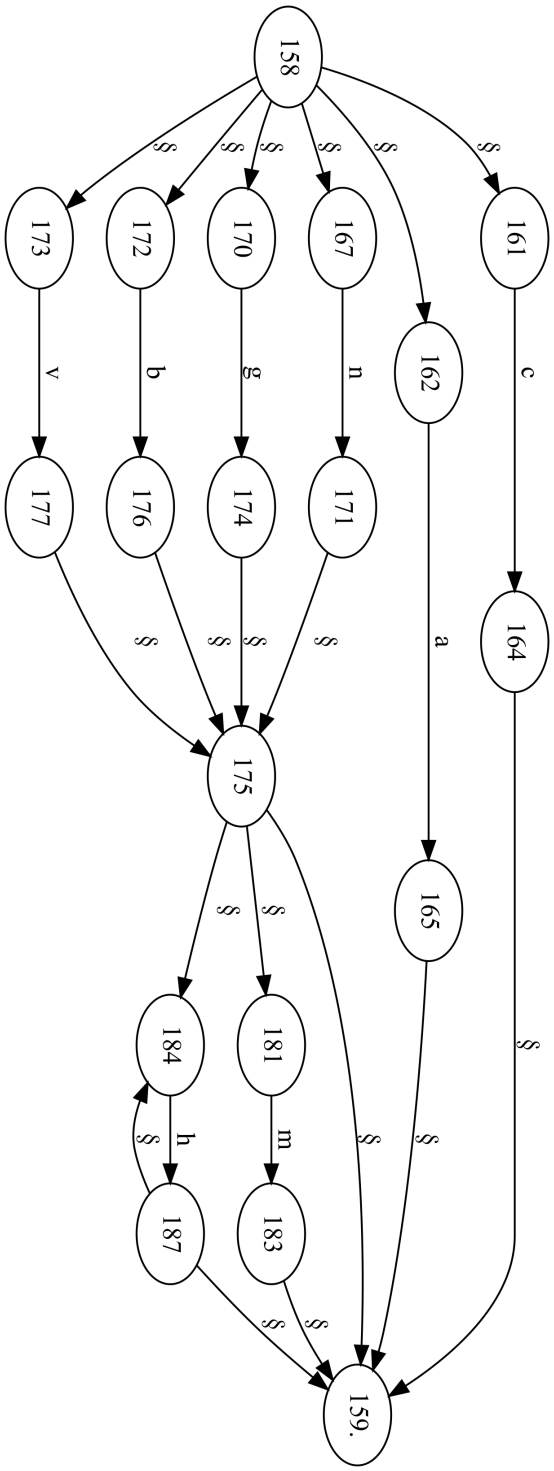


Figure 7.9: Formal method, example 1.

The transformation algorithm implemented in the program handles the $a|(b|v|g|nm|h*)|$ regular expression as the $a|((b|v|g|n)(m|h*))|$. The two regular expressions are equivalent, and if transformed by the transformation algorithm, result in the same NFSM diagram depicted in figure 7.9.
Applying the formal method to determine the bound, results in maximum six NFSM copies, which can be easily verified by examination of the NFSM diagram (figure 7.9). When performing string matching in the program, the run-time number of NFSM copies is printed in the output window. This is another mechanism to verify correctness of the formal method developed and implemented in this work. Around dozen of strings have been matched against the $a|(b|v|g|nm|h*)|$ regular expression, and maximum number of NFSM copies have not exceeded six.
The second example is for the $a|(b|v|g|(nm)|h*)|$ regular expression. The corresponding NFSM diagram is depicted in figure 7.10.

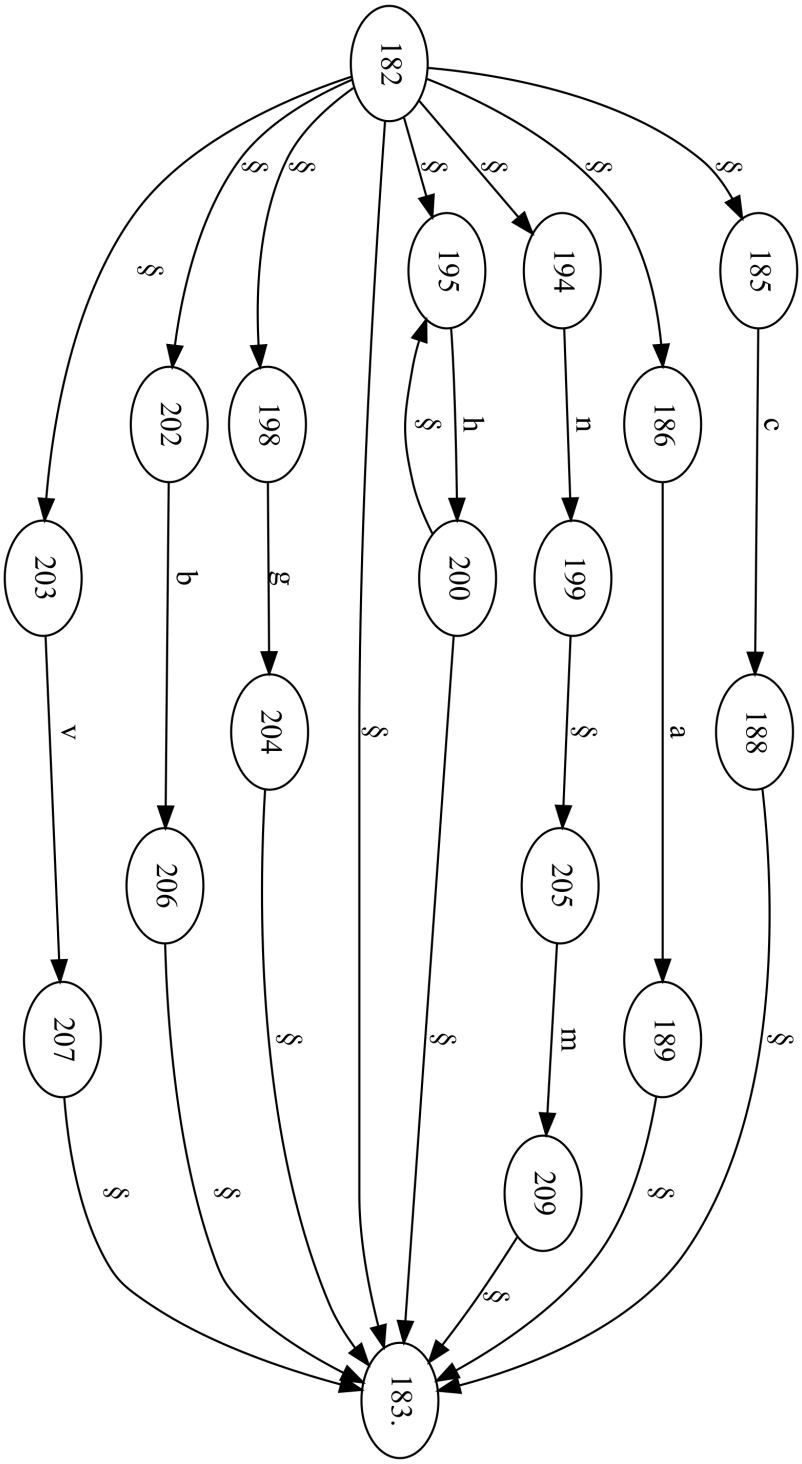


Figure 7.10: Formal method, example 2.

For the NFSM in the second example, the formal method computes the bound to be seven, which can be verified by examination of the diagram depicted in figure 7.10.
The third example demonstrates that the bound computation algorithm successfully escapes local maximums and finds correctly the global maximum. For this example the $((a|b)c)|((d|e)f)(g|m|n|k|l|p|z)$ regular expression is used, which is depicted in figure 7.11.
The bound computation algorithm gives the value seven as the bound for this NFSM, which is correct as can be seen in the NFSM diagram (figure 7.11). As is stated above, the bound computation algorithm starts from the initial state of an NFSM. In the case of the NFSM depicted in figure 7.11, the first bound it finds equals four, but it is not the global maximum.

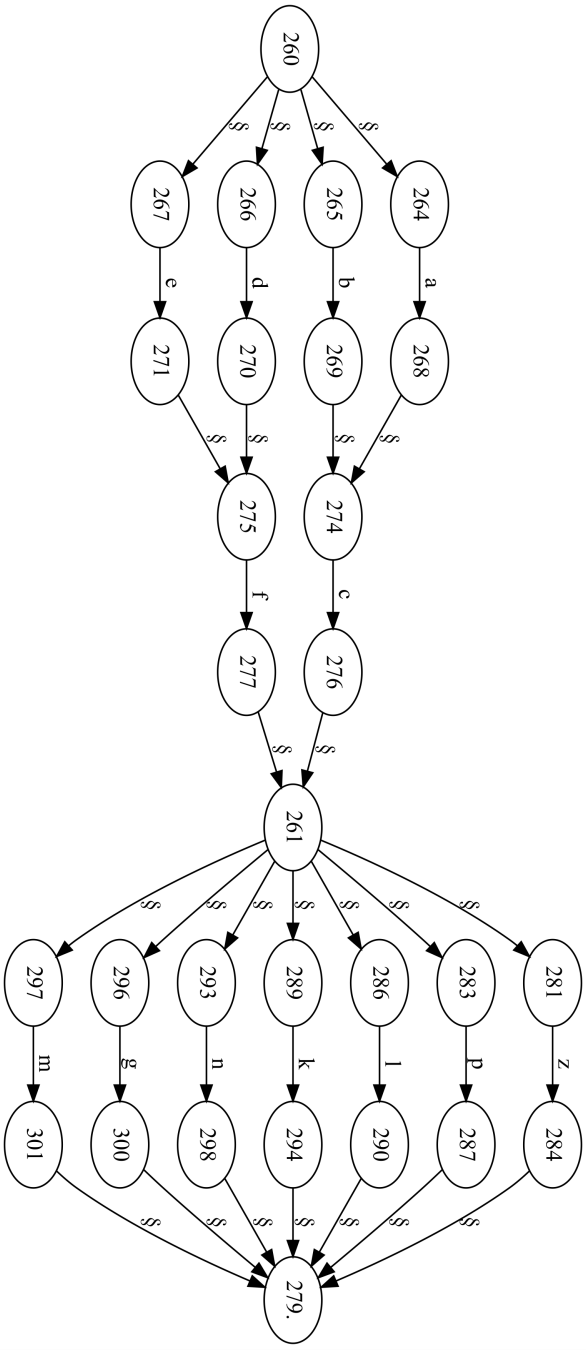


Figure 7.11: Formal method, example 3. Escaping local maximums.

The following example demonstrates the limitations of the bound computation algotirhm. Let us consider the NFSM depicted in figure 7.12. The NFSM is constructed for the $((a|b|c*|d|e)(fu|(mt|d)+|(x|j|f|n|b)*(k*t|p+|o|e)|y)?)|((g|h+|m|n|(o|z|d|a|b))(w(k|d+|y?)|z(fn|h|u)|t|q*|e))$ regular expression.
Running the formal bound computation algorithm on the NFSM gives 21 copies as a result. Closely examining the NFSM diagram (figure 7.12), reveals that the actual bound is lower than is computed. Starting from the initial state 1566 the following 15 states are reached by the §-transitions (also known as lambda-transitions): 1592, 1593, 1609, 1571, 1575, 1591, etc. This big OR-subgraph converges to the two states 1584, and 1623. We can divide the this OR-subgraph into two smaller OR-subgraphs: one converging to the state 1584, the other to 1623. The two OR-subgraphs have common transition symbols "b", and "d", which meads that two NFSM copies, one with the current state 1584, the other with 1623, are left after reading, for example, symbol "d". But after that, there are tree possibilities: symbol "f", or symbol "w", or some other symbol is read. If symbol "f" is read, the NFSM copy with the current state 1623 is deleted, the NFSM copy with the current state 1584 performs the transition to the state 1620. From the state 1620 , 13 states are reachable. If symbol "w" is read, only NFSM copy with the current state 1623 is left, and it performs the transition to the state 1629. From the state 1629, 8 sates are reachable. If other than "f" and "w" symbol is read, then all NFSM copies are deleted. The real bound is 15 copies.
It is obvious that the bound computation algorithm is unable to find the mutual exclusion of the two NFSM copies left after the big OR-subgraph.
The improved version of the bound computation algorithm has been developed, which is able to track NFSM copies for mutual exclusion. For example, it is impossible, considering the execution rules, to have an NFSM copy with the current state 1620, and an NFSM copy with the current sate 1629 in the NFSM diagram depicted in figure 7.12.
The improved bound computation algorithm has the following steps:

1. Starting from an initial state, perform all possible §-transitions (for all NFSM arrays, in case there is more that one array). On each reached state create an NFSM copy with the current state, which equals to the reached state. Newly created NFSMs are put into the array for which the processing is performed.

2. Record the number of reached states in the step 1 for each array of NFSMs individually. Because the arrays correspond to the symbols from the set created in the step 3, the maximum number of reached states for each array corresponds to the specific paths taken in the NFSM diagram with the consideration of possible mutual exclusion of NFSM copies. At this point, all the reached states have only non-§-transitions.

3. Create a set of symbols, which contain the symbols of the transitions of the states reached in the step 1 (It is created each time anew).

4. For each symbol of the set, make a copy of the NFSMs that have been created in the step 1, and feed a symbol to the NFSMs. After the symbol is read, the NFSMs perform transitions for the symbol and reach new current states (new NFSMs created on non-deterministic transitions). Those NFSMs with current states which have no specified transitions for the symbol read are deleted.

5. Repeat from the step 1 for each NFSMs array (array for each symbol of the set created in the step 3) until all arrays are empty.

The improved bound computation algorithm has been implemented in software and tested. It shows correct results for the above examples.

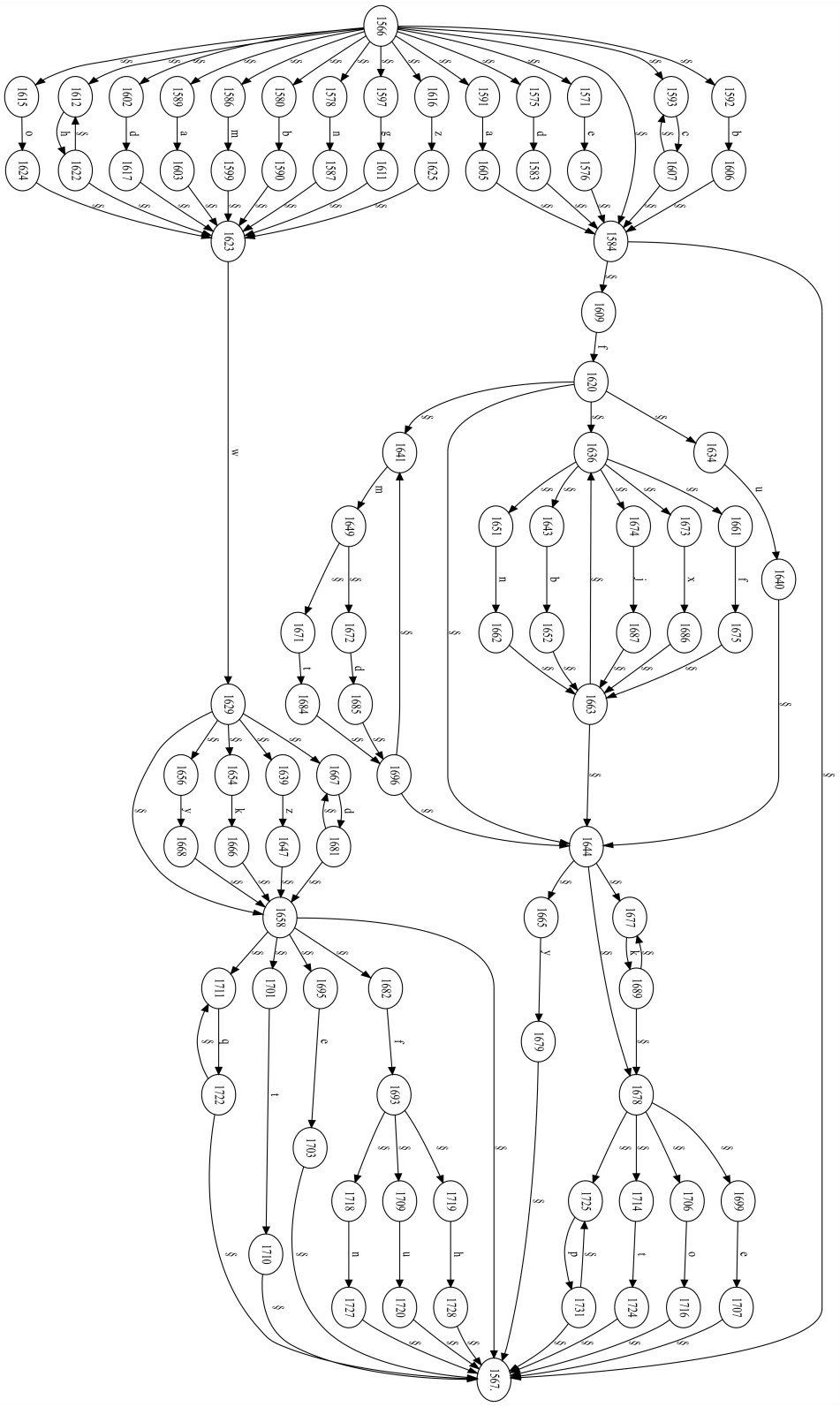


Figure 7.12: Formal method, example 4. Limitations of the bound computation algorithm.

Chapter 8

Architecture of the Computer Program

The computer program for converting regular expressions to NFSMs and computing bounds on number of NFSM copies is developed in this work. Copies of an NFSM are created when nondeterministic transitions are performed in execution. Since practical interest of NFSMs is in their implementation on an FPGA for fast string matching [1] or meta-computation [13], computing bounds on maximum number of copies of an NFSM, further on just "bounds", is required to stay withing limited resources of an FPGA.
The program is developed in Microsoft Visual Studio Community 2015 using C++ language and Microsoft Foundation Classes (MFC). The program has the Graphical User Interface (GUI) depicted in figure 7.1. During the work, a great amount of information and guidelines is accessed on the Microsoft Developer Network (MSDN) [14]. In the implementation of the program, the Standard Template Library (STL) is extensively used. The STL is a software library which offers tested, well designed and implemented, the most commonly used containers and algorithms. Information on how to use STL and examples have been accessed on the website [15]. By containers in STL are meant storage elements like arrays, linked lists, sets and others. Algorithms in STL perform operations on the containers like searching, sorting, etc.
The program code is divided into separate files. The idea is to write all the algorithms created in this work to be cross-platform and put them into separate files containing only platform-independent code. These files named "nfsm.cpp" and "nfsm.h". Other source-code files contain platform-dependent code using mostly MFC for the GUI.
The debugging of the program is performed using a built into Visual Studio debugger.

8.1 Classes

In this section, the classes used in the program are described. The implementation details are mostly omitted, since the code is self describing and easy to follow. Some parts of the implementation are described, as author sees them rather vague.

8.1.1 NFSM Class

The NFSM class represents a complete NFSM. It has the following data members:

- *m_1_structure* contains pairs of a symbol and initial and final states of an NFSM for the regular expression represented by this symbol. This STL container is *map*, which means that if supplied with a symbol, it will return corresponding initial and final states of the NFSM for the symbol.
- *m_2_structure* contains pairs of a string and initial and final states of an NFSM for the regular expression represented by this string. This STL container is also *map* and if supplied with a string, will return the corresponding initial and final states of the NFSM. This container is used to store initial and final states of NFSMs for a symbol followed by one of the metacharacters $*, +, ?$.
- *m_3_structure* contains pairs of a string and initial and final states of an NFSM for the regular expression represented by this string. This is *map* container. Initial and final states of NFSMs for brackets are stored in this container.
- *m_4_structure* contains pairs of a string and initial and final states of an NFSM for the regular expression represented by this string. This is *map* container. In this container initial and final states of an NFSM for the hole regular expression are stored.
- *m_or_structure* contains pairs of a string and initial and final states of an NFSM for the regular expression represented by this string. In this container initial and final states of NFSMs for *OR* regular expressions are stored.
- *m_states* contains objects of *State* class. It is array of size $MAX\_NUMBER\_OF\_STATES$, currently it is 5000. The reason behind fixed size array for the storage of *State* objects is that pointers to them should be valid throughout the construction of NFSM. The STL containers with growing-on-demand size do not guarantee that pointers stay valid after growth of container size.
- *m_current* stores the pointer to the current state of an NFSM. In execution each NFSM can have only one current state.

- *m_constructed* stores a boolean value. If *true*, the NFSM is constructed, otherwise it is not.
- *m_valid* stores a boolean value. If *true*, the NFSM should not be deleted, otherwise it is deleted because it reached the final state when there are still symbols to read or there are no transitions specified for a current state and a symbol read.
- *m_regexpr* stores a regular expression from which the NFSM is constructed.
- *m_output* stores a pointer to the MFC output window. It is used to display output of the program.
- *m_s_id* is an integer representing the ID of a state. It is incremented whenever a new state is created ensuring that all states have unique IDs.
- *m_t_id* is an integer representing the ID of a transition. It is incremented whenever a new transition is created ensuring that all transitions have unique IDs.

The NFSM class has the following methods (functions):

- $NFSM()$ is the default class constructor.
- $NFSM(std :: string\ regexpr, CWnd\ *\ output)$ is a class constructor taking a regular expression and a pointer to the MFC output window.
- $friend\ int\ RUN :: make\_transition(char\ input, bool\ last\_ch)$ is a friend (has access to the private data members of the NFSM class) function which performs transitions for the symbol *input* in NFSMs from a current state. First, this function performs all the §-transitions from a current state making copies of the NFSM on each nondeterministic transition, and then for all available copies of the NFSM it performs transitions for the symbol provided in *input*. After §-transitions are performed again. If an NFSM reaches the final state and *last_ch* is not true, the NFSM is deleted, otherwise the string is accepted. If for a current state of the NFSM there are no transitions defined for the symbol, the NFSM is deleted. The function belongs to the class *RUN*.
- $friend\ int\ run\_lambda(RUN\ *\ obj, bool\ last\_ch)$ performs §-transitions. It is used in *make_transition* function. The function implements a §-transition-cycle avoidance mechanism.
- $std :: string\ read\_or(std :: string :: iterator\ it)$ takes a string iterator *it* which points to the "%" symbol in the regular expression string, and reads and returns a substring of the format "%*number*%", which then can be used to access a corresponding NFSM from *m_or_structure*.

- $std :: string\ read\_bracket(std :: string :: iterator\ it)$ takes a string iterator *it* which points to the "$" symbol in the regular expression string, and reads and returns a substring of the format "$$number$", which then can be used to access a corresponding NFSM from $m\_3\_structure$.
- $int\ construct()$ launches construction of an NFSM.
- $int\ formal\_method()$ launches computation of the bound on copies of an NFSM.
- $void\ write\_nfsm(std :: string\ file\_name)$ writes the structure of a constructed NFSM into the file $file\_name$ using the DOT format. The file is created in the same directory as the executable of the program.
- $StateCouple\ make\_star\_NFSM(State * s\_init, State * s\_final)$ takes as parameters the initial and final states of an NFSM and creates a new NFSM for a symbol or brackets (represented by the passed as parameters NFSM) followed by "*" metacharacter, and returns the initial and final states of the created NFSM. So the function makes the following transformation: NFSM -¿ (NFSM)*. States of the NFSM passed as parameters are copied.
- $StateCouple\ make\_plus\_NFSM(State * s\_init, State * s\_final)$ takes as parameters the initial and final states of an NFSM and creates a new NFSM for a symbol or brackets (represented by the passed as parameters NFSM) followed by "+" metacharacter, and returns the initial and final states of the created NFSM. The function makes the following transformation: NFSM -¿ (NFSM)+. States of an NFSM passed as parameters are copied.
- $StateCouple\ make\_question\_NFSM(State * s\_init, State * s\_final)$ takes as parameters the initial and final states of an NFSM and creates a new NFSM for a symbol or brackets (represented by the passed as parameters NFSM) followed by "?" metacharacter, and returns the initial and final states of the created NFSM. The function makes the following transformation: NFSM -¿ (NFSM)?. States of an NFSM passed as parameters are copied.
- $StateCouple\ make\_or\_NFSM(State * s\_init\_1, State * s\_final\_1, State * s\_init\_2, State * s\_final\_2)$ takes as parameters initial and final states of two NFSMs, NFSM1 and NFSM2, and creates a new NFSM for "$NFSM1|NFSM2$" , and returns the initial and final states of the created NFSM.
- $StateCouple\ make\_simple\_NFSM(State * s\_init, State * s\_final)$ creates an NFSM for brackets not followed by any of the "*", "+", "?" metacharacters, and returns the initial and final states of the created NFSM. States of NFSMs passed as parameters are copied.

- *StateCouple make_one_symbol_NFSM(char ch)* takes as parameter a symbol, creates an NFSM for the symbol, and returns the initial and final states of the created NFSM.

- *StateCouple connect_NFSM(State * s_init_1, State * s_final_1, State * s_init_2, State * s_final_2)* takes initial and final states of two NFSMs, NFSM1 and NFSM2, connects the final state of NFSM1 with the initial state of NFSM2, returns initial and final states of the resulting NFSM. States of NFSMs passed as parameters are copied.

- *StateCouple make_bracket_NFSM(std :: string :: iterator i, std :: wstring & output_ws, char type)* takes as parameters an iterator i pointing to the symbol ")" in the regular expression string, reference to the output string, and a symbol representing the type of an NFSM to be created ("*", "+", "?"), creates an NFSM for brackets, and returns initial and final states of the created NFSM. This function uses *make_star_NFSM*, *make_plus_NFSM*, *make_simple_NFSM*, or *make_question_NFSM* depending on the *type* parameter value to perform transformations.

- *StateCouple copy_nfsm(State * init, State * final_s)* takes as parameters initial and final states of an NFSM to be copied, copies all the states of the NFSM creating new state IDs, and returns the initial and final states of the copied NFSM.

- *std :: wstring third_it_m()* performs the third iteration of the transformation process. The third iteration creates NFSMs for simple OR regular expressions. The return value is the output of the function to be printed in the "Output" field of the User Interface (UI).

- *std :: wstring fourth_it()* performs the fourth iteration of the transformation process, which creates NFSMs for brackets. The return value is the output of the function to be printed in the "Output" field of the User Interface (UI).

- *std :: wstring fifth_it()* performs the fifth iteration of the transformation process, which creates NFSMs for complex OR regular expressions. The return value is the output of the function to be printed in the "Output" field of the User Interface (UI).

- *void set_invalid()* sets an NFSM to the invalid state, meaning that it should be deleted.

- *bool is_valid()* checks whether an NFSM is valid, and returns boolean true if it is valid, otherwise it returns false.

- *void optimize()* performs optimization on the created NFSM. Optimization consists in removing superfluous states. This function is turned on by checking the "Use optimizing transform" checkbox in the UI.

8.1.2 State class

The *State* class represents a state in an NFSM. It has the following data members.

- $std :: vector < Transition > \ m\_out$ contains *Transition* objects that represent out-coming transitions of the *State* object.
- $std :: vector < Transition > \ m\_in$ contains *Transition* objects that represent in-coming transitions of the *State* object.
- $int \ m_id$ unique ID of *State* object.
- $bool \ m\_init\_state$. If true, the *State* object is the initial state of an NFSM.
- $bool \ m\_final\_state$. If true, the *State* object is the final state of an NFSM.
- $bool \ m_empty$. If true, the *State* object is not initialized.

The *State* class has the following methods.

- $State()$ is the default constructor of the *State* class.
- $State(int \ id, bool \ initial = false, bool \ finals = false)$ is a custom constructor of the *State* class. The first parameter is the unique ID of the *State* object, the second parameter which has the default boolean value false, specifies if the *State* object is the initial state of an NFSM, the third parameter with the default boolean value false, specifies if the *State* object is the final state of an NFSM.
- $friend \ int \ RUN :: make\_transition(char \ input, bool \ last\_ch)$ is a friend function, meaning that it has access to all the private members of the *State* class. The function is used in execution of an NFSM. It performs transitions for a symbol, passed in the *input* parameter, from a current sate. If after performing transitions, the final state is reached and the *last_ch* is false, the NFSM is deleted, otherwise, string is accepted. The *last_ch* is true if the symbol passed in the *input* parameter is the last symbol of the string.
- $friend \ int \ run\_lambda(RUN \ * \ obj, bool \ last\_ch)$ has already been described in the NFSM class methods.
- $void \ set\_transition(int \ id, char \ symbol, State \ * \ in)$ sets up a transition for the *State* object. The *id* parameter is the unique ID of the *Transition* object, the *symbol* parameter is the input symbol on which the transition is performed, the *in* parameter is the pointer to a *State* object to which the transition leads.
- $void \ set\_final(bool \ b)$ sets the *State* object as final state of an NFSM.
- $void \ set\_initial(bool \ b)$ sets the *State* object as initial state of an NFSM.

- *bool is_final*() checks whether the *State* object is the final state of an NFSM, and returns boolean true if it is final, otherwise it returns false.
- *bool is_initial*() checks whether the *State* object is the initial state of an NFSM, and returns boolean true if it is initial, otherwise it returns false.
- *void set_id*(*int id*) sets the ID of the *State* object.

8.1.3 Transition class

The *Transition* class represents transitions of states. It has the following data members.

- *int m_id* is a unique ID of the *Transition* object.
- $State * m\_from$ is a pointer to a *State* object, from which the transition starts.
- $State * m\_in$ is a pointer to a *State* object, to which transition leads.
- *char m_symbol* is an input symbol on which the transition is triggered.

The *Transition* class has the following methods.

- $Transition()$ is the default constructor of the *Transition* object.
- $Transition(int\ id, char\ symbol, State * from, State * in)$ is the custom constructor of the *Transition* object. The *id* parameter is a unique ID of the *Transition* object, the *symbol* parameter is a symbol on which the transition is triggered, the *from* parameter is a pointer to *State* object from which the transition starts, the *in* is a pointer to *State* object to which the transition leads.
- *int get_id*() returns the unique ID of a *Transition* object.
- $State * get\_from\_state()$ returns a pointer to a *State* object from which the transition starts.
- $State * get\_in\_state()$ returns a pointer to a *State* object to which the transition leads.

8.1.4 RUN class

The *RUN* class represents an execution of an NFSM. It has the following member functions.

- $RUN(NFSM\ machine)$ is the custom constructor of the class.

- *int make_transition(char input, bool last_ch)* performs a transition for the input symbol passed in *input* parameter. If the *last_ch* parameter is false and after the transition an NFSM reaches the final state, the NFSM is deleted, otherwise the complete string is accepted. The return values are:
  - $VALID\_TRANSITION$ means that a valid transition has been performed.
  - $INVALID\_TRANSITION$ means that an invalid transition was encountered, the NFSM is deleted.
  - $FINAL\_STATE$ means that the final state has been reached.
  - $NON\_D\_TRANSITION$ means that a non-deterministic transition has been performed.
- $friend\ int\ run\_lambda(RUN\ *\ obj, bool\ last\_ch)$ described in the subsection "Helper functions".

The *RUN* class has the following data members.

- $std :: vector < NFSM >\ m\_nfsms$ contains copies of an NFSM created while execution.
- $CWnd\ *\ m\_output$ is a pointer to the MFC's *CWnd* object, which represents the output window in the GUI.

8.2 Helper Functions

The following helper functions are used in the program source code:

- *bool is_meta_char(char ch)* takes as an argument a character and returns boolean true if the character equals to one of the characters in the set {"(",")","+","*","?",".","|"}.
- $State\ *\ find\_initial(std :: vector < State >\ *)$ takes as a parameter a vector of pointers to *State* objects, finds among them one that represents the initial state of an NFSM (*m_init_state* is true) and returns a pointer to it.
- $State\ *\ find\_final(std :: vector < State >\ *)$ takes as a parameter a vector of pointers to *State* objects, finds among them one that represents the final state of an NFSM (m_final_state is true) and returns a pointer to it.
- *bool is_numeric(char ch)* takes as a parameter a character, and returns boolean true if the character is a digit (0, 1, 2, 3, ..., 9).
- *bool is_bracket(char ch)* takes as a parameter a character, and returns boolean true if the character equals "(" or ")".

- *bool is_meta_char_nb(char ch)* returns true if the character passed in the *ch* parameter equals to one of the set {" + "," ∗ ","?",".","|"}.
- *bool is_star_plus_quest(char ch)* returns boolean true if the character passed as a parameter equals to "+" or "*" or "?".

8.3 Class diagram

The class diagram is depicted in figure 8.1. There are four classes: *RUN*, *NFSM*, *State*, *Transition*, *RUN*. From the Class diagram depicted in figure 8.1, it can be seen that the following relations exist between the classes: *RUN* contains *NFSM*, *NFSM* contains *State*, *State* contains *Transition*.

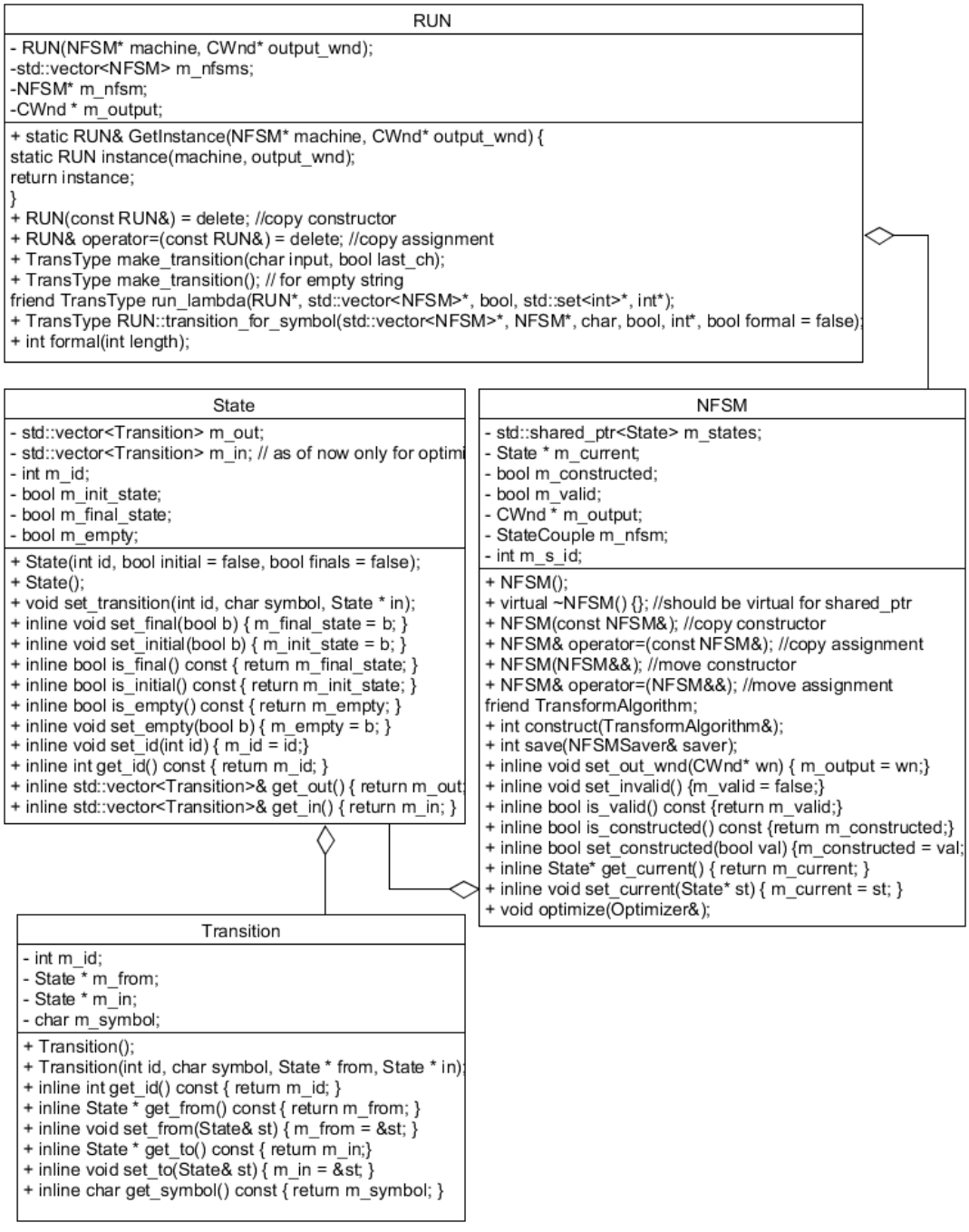


Figure 8.1: Class diagram.

The class diagram of the Thompson's transformation algorithm implementation is depicted in figure 8.2. The NFSM's *construct(TransformAlgorithm&)* method takes as a parameter a reference to the *TransformAlgorithm* abstract class. The *TransformAlgorithm* abstract class provides an interface, so that any transformation algorithm which implements this interface can be used by the NFSM's *construct(TransformAlgorithm&)* method. This allows for an easy addition of other transformation algorithms, maintaining modular structure of the overall design.
The same approach is taken in the implementation of the optimization and saving algorithms.

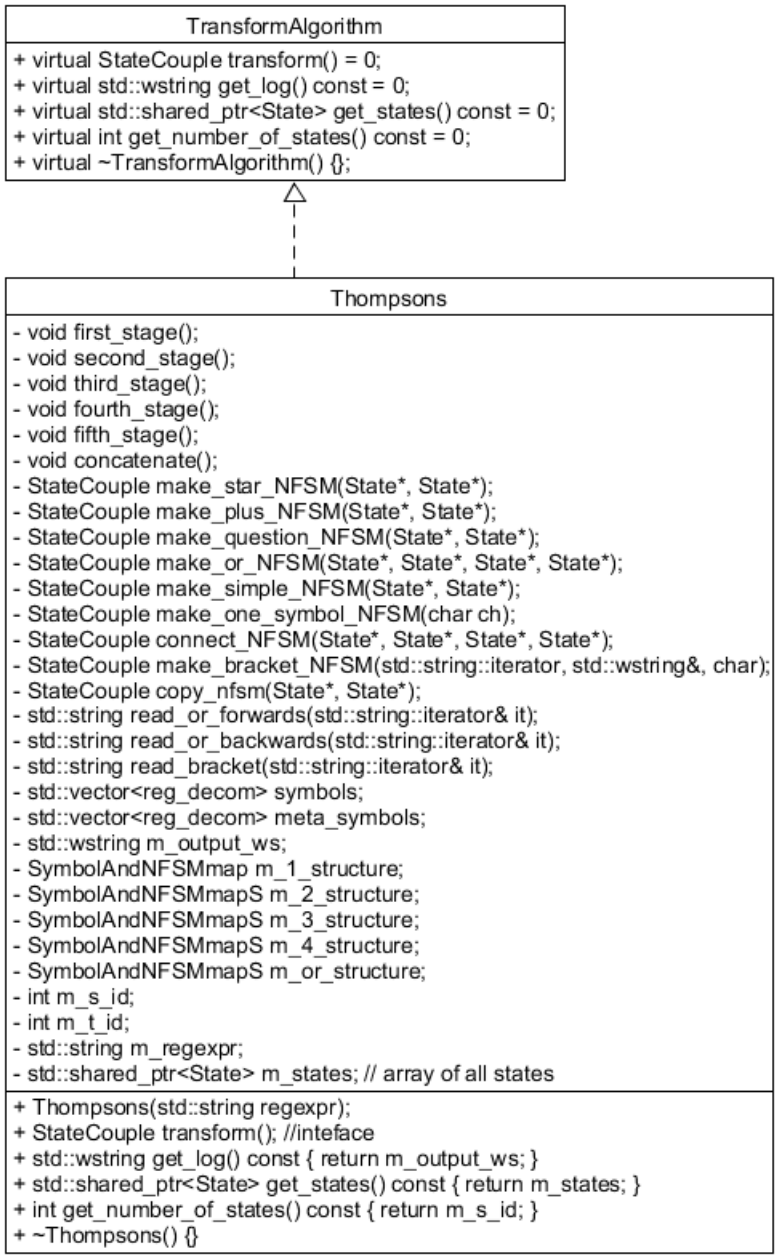


Figure 8.2: Class diagram of the Thompson's transformation algorithm implementation.

The class diagrams of the NFSM saving and optimizing algorithms implemen-

tation are depicted in figure 8.3. In the same figure, the class diagrams of the logger and the custom exceptions are depicted.

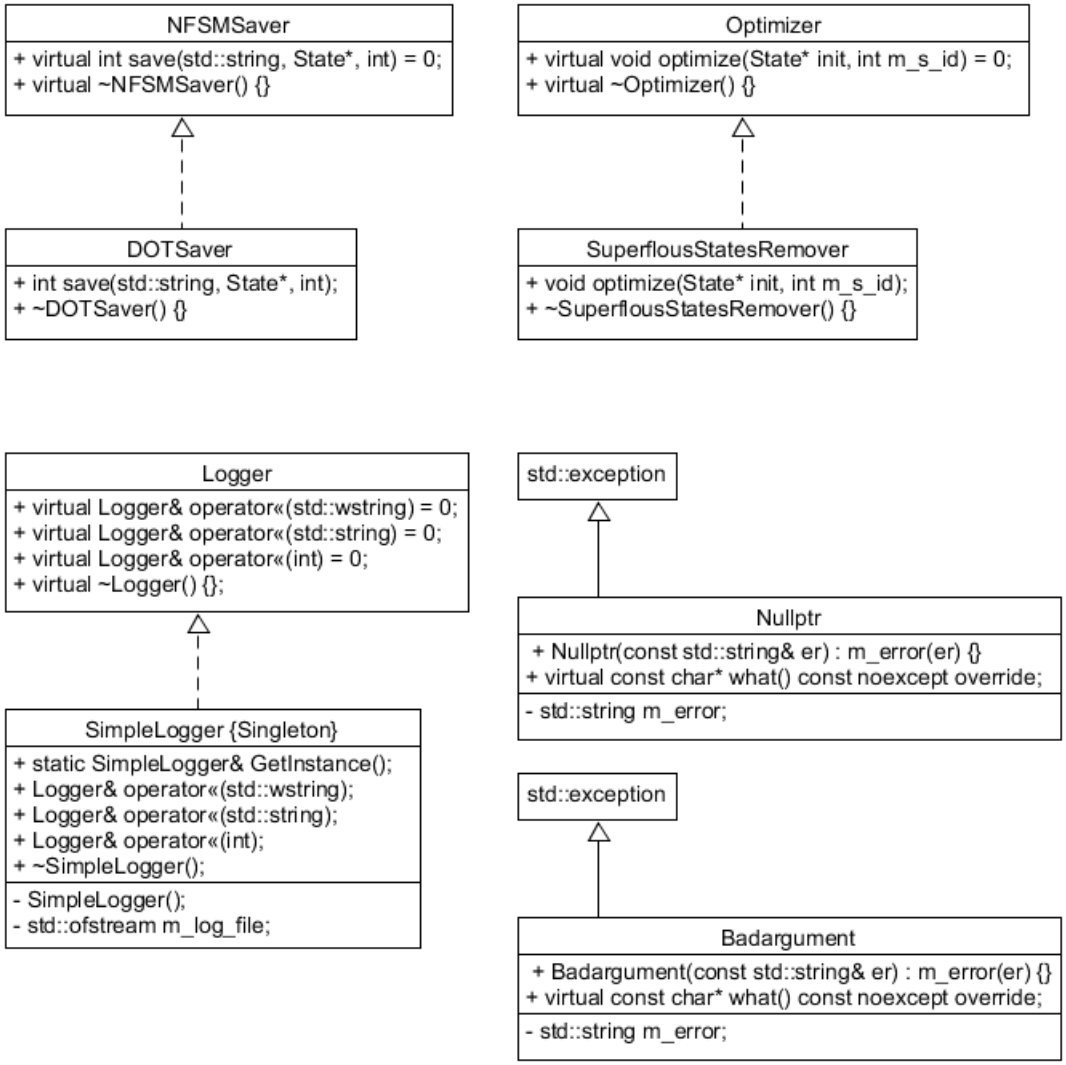


Figure 8.3: Class diagrams of the NFSM saving and optimizing algorithms implementation, as well as class diagrams of the logger and the custom exceptions.

8.4 Run-time Object Diagram

During the design process of the software, considerable attention has been given to the problem of how NFSM objects, which are identical copies except the current state (which is a pointer to a *State* object), should be copied: either copy all the *State* objects or share them. Copying all *State* objects when copying an *NFSM* object is expensive in terms of memory and performance. Because of this performance considerations, it has been chosen to share *State* objects between *NFSM* objects. This design decision has its benefits in that creating a copy of an *NFSM* object is fast, but care should be taken not to corrupt shared *State* objects. During the programming phase, the most trickiest corruption of shared *State* objects was erroneous manipulation of transitions, which are pointers, between *State* objects. The run-time object diagram is depicted in figure 8.4. There is only one object of *RUN* class. According to [16], the *RUN* class is a *singletone*. *Singletone* class can have only one object present. To ensure this, special arrangements needs to be performed. In C++ this can be done by making the constructor of a *singletone* class *private*. This ensures that a *signeltone* object cannot be copied or created more that once [16]. Considerable amount of advice on the C++ software design has been taken from [17] and [18]. The *RUN* object can have many copies of *NFSM* object. The copies of *NFSM* object are identical and differ only by a current state they are in. *NFSM* object has many *State* objects, which in turn, can have many *Transition* objects.

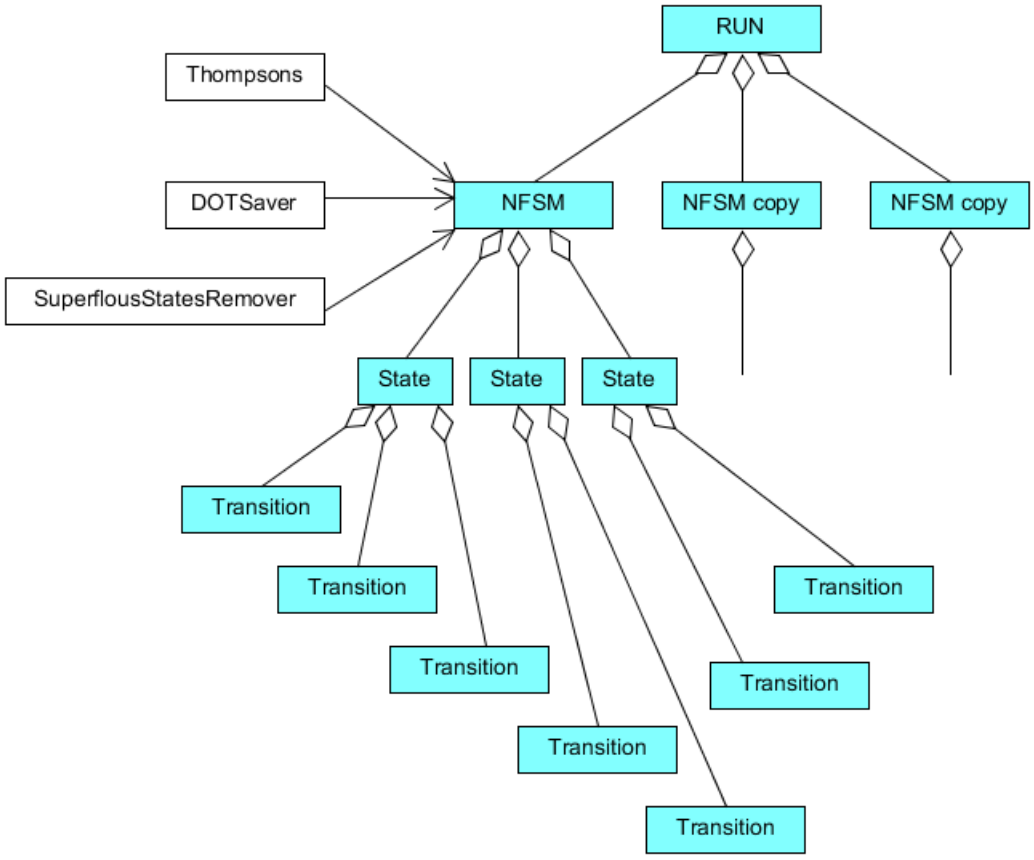


Figure 8.4: Object diagram.

Chapter 9

Results

In this chapter the results achieved in the thesis are presented in a concise form. Starting from the "Theory and Background" chapter, the work naturally progresses through development of the algorithmic solutions for the transformation and bound computation algorithms to the implementation of the developed algorithms in software. The theoretical work in the thesis is represented mainly in the "Theory and Background", and "Computing bounds for NFSM execution" chapters. This includes the research on NFSMs and DFSMs, and their transformation algorithms, proving by logical induction that NFSMs constructed by the adapted Thompson's algorithm always have a finite bound when executed in accordance with the execution rules presented in the "Computing bounds for NFSM execution" chapter. The practical part of the thesis is in development of the software program. The development included many iterations of design and implementation phases to improve code quality. The quality here is measured in modularity, code reuse, error handling, and maintainability. The measurements are relative to the previous version of the code. If the mentioned parameters are higher than ones of the previous version, the new version is accepted. This way, the first working ad-hoc version of the program is improved by increasing modularization and code reuse. After addition of "nice-to-have" functionality like logging, error checking and optimizations, the code size is not increased. For testing purposes and for ease of an NFSM reading, the graphical representation of NFSMs is implemented using the open source software GraphViz. The developed software program transforms a regular expression to the corresponding NFSM, and performs the bound computation. The NFSM is saved in a DOT-format file, which can be read and visualized by GraphWiz. The final version of the program is tested with regular expressions of different complexities designed to perform full code coverage.

Chapter 10

Discussion

In this chapter results that have been achieved in the thesis are discussed.
After research on NFSMs and DFSMs, it has been noted that an NFSM implements a regular expression with less states than an DFSM, and offers speed-up by parallel execution. The mentioned speed-up is theoretical and haven't been proved by running benchmarks. It is obvious that the speed-up is strongly dependent on implementation, and that there should inevitably be performance penalties of managing parallel execution. Some analogy can be drawn to managing threads in an operating system: time required to create a thread, and if the execution time of a thread is comparable to the creation time, there is no speed-up, but speed-down. So the same principle applies to NFSMs. If an NFSM is small, and it's corresponding DFSM's execution time is comparable to the time required to create a copy of the NFSM, there is no speed-up.
The other, apart from simple substring matching, usage of NFSMs is in meta-computation. The concept, as to my knowledge, is rather informally presented in the literature. This concept also relates to a control logic being implemented as an NFSM. The idea is quite simple: to each subexpression in a regular expression an action is attached. For example, consider the regular expression $(ab)@2(h|p)@5$. The regular expression is composed of two subexpressions (ab) and $(h|p)$ to which actions are attached. Here the @ symbol is used to denote an action, and the number after it is the ID of an action, which can be just a function. If the "ab" substring is matched, then the action with the ID 2 is performed, which can take arguments "a" and/or "b", and perform some action on them. The practical value of such meta-computation is a question. But due to possible application of such technique, I left possibility in the architecture of the program to extend it for meta-computations. The regular expression syntax implemented does not allow digits to be a simple symbol, neither they are meta-characters. They used by the transformation algorithm for subexpression substitution. For example, $\ldots|\ldots$ is substituted with $\%number\%$, and $(\ldots)$ is substituted by $\$number\$$. It is easy to modify the transformation algorithm, so that it can handle actions attached to subexpressions, where a number represents a function ID. The ID of a function can be a number of array element which is

a function pointer.
The more practical use of NFSMs on an FPGA is for network traffic monitoring. It can be used for statistic purposes or to prevent hacker attacks.
During the work, it is found that NFSMs created by the adapted Thompson's algorithm are regular in structure. They are composed of limited number of sub-NFSMs which are connected together in a limited number of ways. The regular structure together with the execution rules guaranties that NFSMs constructed that way always have finite bound on number of copies when executing.
The software program developed in this thesis proved to be stable handling complex regular expressions. The formal method for computing bounds is in agreement with test runs of NFSMs. The speed of the algorithm implementation is within 5 seconds for a maximum length regular expression on a medium class laptop computer.

Chapter 11

Conclusion

In this work NFSMs and DFSMs have been studied. It has been shown, that NFSMs constructed by the adapted Thompson's algorithm and executed following the execution rules, always have a finite bound on copies, which is not dependent on input. The Thompson's transformation algorithm has been adapted and implemented in software. The bound computation algorithm has been developed and implemented in software based on reachability of states in an NFSM state diagram, which is directed cyclic graph. The software solution has been tested and showed consistent results. To verify the correctness of the transformation, the visualization of an NFSM state diagram has been employed. The free and open-source GraphViz v. 2.38 is used for visualization of NFSM diagrams. The software program, developed in this thesis, saves structure of a constructed NFSM in a DOT-format file which is read by GraphViz.
Other algorithmic solutions concerning NFSMs, developed and implemented during the work on the thesis, are following:

- A regular expression syntax checking. Numbers are not allowed, but not checked (they are reserved for actions performed on subexpressions, implementation of which is left for the future work).
- An optimization algorithm for removing superfluous states from a constructed NFSM. It does not change functionality, but makes it easier to view NFSM diagrams.
- User-friendly user interface.

11.1 Future work

For future work it is left to implement actions on subexpressions, a good syntax for this is introduced in [19].
The other possibility for improvement is in extending functionality of the program to perform string and sub-string matching on input data from files or

other sources. As of now it accepts a string and matches it entirely and not substrings of it. The matched substrings then could be saved in a database for other manipulations.

Bibliography

[1] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," in *Journal of Signal Processing Systems, Volume 51, Issue 1*, pp. 99–121, Springer US, 2008.

[2] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. USA: Addison-Wesley, 2 ed., p. XIV, 513, 2001.

[3] Y. Su and S. Y. Yan, *Principles of Compilers. A New Approach to Compilers Including the Algebraic Method*. Germany: Springer, p. 451, 2011.

[4] M. Pilgrim, *Dive Into Python 3*, ch. Regular Expressions, pp. 69–85. Berkeley, CA: Apress, 2009.

[5] P. Linz, *An Introduction to Formal Languages and Automata*. USA: Jones and Bartlett publishers, 3 ed., p. XI, 410, 2001.

[6] O. Boldt and H. Jürgensen, *Automata Implementation, 4th International-Workshop on Implementing Automata, WIA'99 Potsdam, Germany, July 17-19, 1999*. Germany: Springer, p. 183, 1999.

[7] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions, *POSIX basic regular expressions*. Accessed 2016.

[8] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions, *POSIX extended regular expressions*. Accessed 2016.

[9] http://pubs.opengroup.org/onlinepubs/9699919799/, *Regular Expression Definitions*. Accessed 2016.

[10] K. Thompson, *Programming Techniques: Regular expression search algorithm*. Communications of ACM, Vollume 11, issue 6, pp. 419-422, 1968.

[11] https://en.wikipedia.org/wiki/Thompson%27s_construction, *Thompson's construction*. Accessed 2016.

[12] http://www.graphviz.org/, *GraphViz official web-site*. Accessed 2016.

[13] R. Sidhu and V. K. Prasanna, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream: 12th International Conference, FPL 2002 Montpellier, France, September 2–4, 2002 Proceedings*, ch. Efficient Metacomputation Using Self-Reconfiguration, pp. 698–709. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[14] https://msdn.microsoft.com/de-de/default.aspx, *Microsoft Developer Network web-site*. Accessed 2016.

[15] http://www.cplusplus.com, *C++ and STL reference web-site*. Accessed 2016.

[16] M. Zandstra, *PHP Objects, Patterns, and Practice*, ch. Patterns for Flexible Object Programming, pp. 175–194. Berkeley, CA: Apress, 2013.

[17] B. Stroustrup, *The C++ Programming Language 4th Edition*. Addison-Wesley, p. 1267, 2013.

[18] M. Gregoire, N. Solter, and S. Kleper, *Professional C++*. John Wiley and Sons, Inc., p. 1053, 2011.

[19] K. Volden, *Master thesis: Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC*. NTNU, IET, p. 137, 2011.