STATE CHANGE OPERATOR:
ALGEBRAIC AND LANGUAGE CLASS PROPERTIES

A MASTER'S THESIS

in

Computer Engineering

Atılım University

by

AHMET ÇEVİK

JULY 2009

GCRIS

STATE CHANGE OPERATOR: ALGEBRAIC AND LANGUAGE CLASS PROPERTIES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

BY

AHMET ÇEVİK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2009

GCRIS

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

Prof. Dr. Abdurrahim Özgenoğlu
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. K. İbrahim Akman
Head of Department

This is to certify that we have read the thesis "State Change Operator: Algebraic and Language Class Properties" submitted by Ahmet Çevik and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Halit Oğuztüzün
Co-Supervisor

Asst. Prof. Dr. Hürevren Kılıç
Supervisor

Examining Committee Members

Asst. Prof. Dr. Çiğdem Turhan

Asst. Prof. Dr. Osman Abul

Asst. Prof. Dr. Deepti Mishra

Date: 01.07.2009

GCRIS

I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

AHMET ÇEVİK

GCRIS

ABSTRACT

STATE CHANGE OPERATOR:
ALGEBRAIC AND LANGUAGE CLASS PROPERTIES

Çevik, Ahmet
M.S., Department of Computer Engineering
Supervisor : Asst. Prof. Dr. Hürevren Kılıç
Co-Supervisor : Assoc. Prof. Dr. Halit Oğuztüzün

July 2009, 53 pages

A new proper subset of regular languages, inspired from abstract power consumption attribute of computational devices, is introduced in this thesis. This new subset is called state change languages. For our purpose, we define a unary operator whose domain is finite automata and range is state automata. We have proven that the new class is closed under union and intersection, but not under complementation. The decidability of this set is also discussed. Then, the notion of abstract power consumption is given and this notion ,along with the satisfiability of state change languages, is axiomatized in the last section. Finally, the importance of the existence of a universal operator that shows the abstract power consumption characteristic of well defined abstract models of computation to determine the limits of their enegry consumption for particular algorithms, is pointed out.

Keywords: State change languages, finite automata, abstract power consumption, closure properties, decidability, axiomatization.

ÖZ

DURUM DEĞİŞİKLİĞİ İŞLECİ:
CEBİRSEL VE BİÇİMSEL DİL KÜMESİ ÖZELLİKLERİ

Çevik, Ahmet
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Hürevren Kılıç
Ortak Tez Yöneticisi : Doç. Dr. Halit Oğuztüzün

Temmuz 2009, 53 sayfa

Bu tezde soyut hesaplama modellerinin soyut güç tüketimi özelliklerinden esinlenerek düzgün dillerin yeni bir alt kümesi tanımlanmıştır. Bu yeni kümeye durum değişikliği dili adı verilmiştir. Bunun için, tanım kümesi sonlu özdevinir ve erim kümesi durum özdevinir olan birli işleç tanımlanmıştır. Bahsedilen yeni kümenin birleşme ve kesişme özellikleri altında kapalı olduğu, tümleme özelliği altında ise kapalı olmadığı kanıtlanmıştır. Kümenin karar verilebilirliği hakkında tartışılmıştır. Ayrıca soyut güç tüketimi kavramı formal olarak tanımlanmış, ve son bölümde bu kavram durum değişikliği dillerinin sağlanabilirliği ile beraber belitleştirilmiştir. Son olarak, belirli algoritmalar için düzenli olarak tanımlanmış soyut hesaplama modellerinin güç tüketimininin alt limitinin belirlenmesindeki evrensel bir işlecin varlığının önemi vurgulanmıştır.

Anahtar Kelimeler: Durum değişikliği dilleri, sonlu özdevinirler, soyut güç tüketimi, kapalılık özellikleri, karar verilebilirlik, belitleştirme.

GCRIS

To My Family

ACKNOWLEDGMENTS

My very special thanks from my deep heart go to my thesis advisor Hürevren Kılıç. I would not complete my work without his support and motivation. I feel very lucky that I have a mentor like him. I also want to thank him for teaching me Theory of Computation which played an important role in my thesis, and will continue to play in my further academic study. I want to thank my co-advisor Halit Oğuztüzün for his comments and discussions throughout my thesis study. I should not forget to mention my friends for their motivation. Last, but not least, I want to thank my family for their continuous support not only in my studies but in so many other things.

GCRIS

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

DEDICATION vi

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF FIGURES x

CHAPTERS

1 INTRODUCTION 1

2 FINITE AUTOMATA AND REGULAR LANGUAGES 3

2.1 Finite Automata 3

2.2 Regular Expressions 4

2.3 Algebraic Properties of Regular Expressions 8

3 STATE CHANGE LANGUAGES 11

3.1 State Change Operator and State Automata 12

3.2 State Change Languages and Expressions 15

4 CLOSURE PROPERTIES 17

4.1 State Automata with ϵ-Transitions 19

5 DECIDABILITY OF STATE CHANGE LANGUAGES 23

5.1 Emptiness Checking 25

5.2 Language of the Turing machine 26

5.3 An Undecidable Problem 27

6 ABSTRACT POWER CONSUMPTION 30

6.1 Abstract Power and Complexity 30

6.2 An Uncomputable Number 31
7 AXIOMATIZATION OF ABSTRACT POWER CONSUMPTION . . 34
7.1 Logical Description of SCL 34
7.2 Formulas, Models, and Theories 35
7.3 The Axioms 39
7.4 Axiomatized Formal Theories 42
7.5 Satisfiability of Finite Automata Properties 46
8 CONCLUSIONS 49
REFERENCES 51

GCRIS

LIST OF FIGURES

FIGURES

Figure 2.1 The basis of the construction of an automaton from a regular expression. Epsilon moves are indicated as 'eps'. 7

Figure 2.2 Induction to construct finite automaton from a regular expression. . 8

Figure 3.1 A finite automaton A. 13

Figure 3.2 State automaton $B = C(A)$. 13

Figure 3.3 Relationship among language classes involving SCL. 16

Figure 4.1 State automaton S_N recognizing the language $L \circ M$. 21

Figure 4.2 State automaton S' recognizing the language $(L(S))^*$. 21

Figure 5.1 Reduction from L_1 to L_2. The solution of L_2 is the solution of L_1. . 27

GCRIS

CHAPTER 1

INTRODUCTION

Models of computation are mathematical abstractions of computational devices. They are usually used to study the limits of computation. The same models, on the other hand, can also be used to elaborate on related run-time attributes of computational devices. Specifically, power consumption is such an attribute that can be associated with execution of any computing device. Configuration changes during computation can be thought as the main driver behind such consumption.

Power consumption characterization realized by working on abstract computational models rather than on their implementations gives us: (i) an idea about power consumption related limits of their implementations under certain conditions (ii) an abstraction to investigate possible robust classes of languages that can be identified through the very basic idea of power consumption [1]. This in turn may lead to identification of new abstract machines. Power consumption based formalization of digital computing systems has been investigated in [2]. Related to item (ii), search for robust classes of regular languages is known to be an old problem of theory of computing, automata theory, formal verification and regular model checking [4][3]. In [3] for example, a robust decidable class of regular languages recognized by finite ordered monoids, called W, has been proposed. Other example subclasses of regular languages identified due to the "negative result that regular languages cannot be inferred from positive data only" [5] include: k-reversible languages [6], strictly regular languages [7], regular code languages [8] and uniquely terminating regular languages [9].

In this thesis, a simple operator that enables model level power consumption charac-

terization of its possible implementations is proposed. Investigate about a possible robust language class defined by the range description of the mapping is given. For the time being, we restrict ourselves to a unary mapping that operates on state change behavior of finite automata model of computation. As a consequence, a new formal language set what we call State Change Languages (SCL) is identified. Then, the language class and closure properties of SCL are investigated. The reason behind choosing to work on a finite automata instead of transducers [19][20] is to exploit on the recognized language set. Note that the effort can clearly be extended to alternative computational models by defining alternative operators.

The remainder of the thesis has the following organization. In Section 2, finite automata and regular languages are reviewed. In Section 3, the unary state change operator, and the set SCL, obtained by the mapping, are defined. In the same section, it has been shown that SCL is a proper subset of regular languages, and the algebraic properties of the C operator have been discussed. In Section 4, the language properties of SCL including closure and decidability are given. In Section 5 the notion of abstract power consumption for finite automata is discussed shortly. In Section 6, the focus is on the logical perspective where we give an axiomatization of abstract power consumption over finite automata, and use this formalism in the satisfiability of finite automata properties, when given a formal specification. The last section includes the conclusions and future work.

CHAPTER 2

FINITE AUTOMATA AND REGULAR LANGUAGES

A model of computation is a mathematical abstraction of any computational device including todays computers. Although they are not used for any practical reason, they are important to study the limits of computation. In this section, finite automata (FA) and its corresponding language class called regular languages (RL) are formally reviewed in order to give the necessary background to understand the work done. The properties of the language recognized by finite automata are given. Then, we give two known results, as lemmas, which will be used later on.

2.1 Finite Automata

Finite automaton is the simplest model of computation. Informally, a finite state automaton consists of a set of states and a finite control which handles the transitions among the states in response to external inputs. Before giving the formal description, let us define a notation first. For any set Q we write $P(Q)$ to be the *power set* of Q. Now we can write the formal definition of nondeterministic finite automata. We will use, from now on, finite automata and nondeterministic finite automata interchangeably.

Definition 1. A *Non-Deterministic Finite Automata* (NFA) is a 5-tuple
$(Q, \Sigma, \delta, q_0, F)$, where
Q : finite set of states,
Σ : finite set of symbols, called the alphabet,

$\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow P(Q)$, the transition function[1],

$q_0 \in Q$ is the start state, and

$F \subseteq Q$ is the set of accepting states [10].

Definition 2. A language is called *regular* if some finite automaton recognizes it.

Definition 3. Let A and B be two languages. We define the regular operations union, concatenation, and star as follows:

• Union: $A \cup B = \{x | \ x \in A \text{ or } x \in B\}$

• Concatenation: $A \circ B = AB = \{xy | \ x \in A \text{ and } y \in B\}$

• Star: $A^* = \{x_1 x_2 \ldots x_k | \ k \geq 0 \text{ and each } x_i \in A\}$

Regular languages are closed under the operation mentioned above along with intersection, complemenation, difference, reversal, homomorphism and inverse homomorphism. The proof can be seen in [11].

2.2 Regular Expressions

In the previous subsection we focused on abstract state machine description of languages. Now we show an algebraic way to describe regular languages. The notion is called "Regular Expression". Regular expressions (RE) are the declarative way to express strings. It is worth noting that a regular expression E is just an expression, not the language itself. Therefore, we use $L(E)$ to refer the language that E defines. Regular expressions can be defined recursively as follows.

Basis: The basis consists of,

1. ϵ and $\emptyset$ are regular expressions, exhibiting the languages $\{\epsilon\}$ and $\emptyset$ respectively. In other words, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.
2. If $a \in \Sigma$, then a is a regular expression. It should be clear that $L(a) = \{a\}$.

Induction: Inductive step consists of several parts.

1. If A and B are regular expressions, then $A + B$ is a regular expression denoting $L(A + B) = L(A) \cup L(B)$.

[1] ϵ is called the *empty string*.

2. If A and B are regular expressions, then AB is a regular expression denoting $L(AB) = L(A)L(B)$.

3. If A is a regular expression, then A^* is a regular expression denoting $L(A^*) = (L(A))^*$.

4. If A is a regular expression, then so is (A) which denotes the same language as A. That is, $L((A)) = L(A)$.

So far we have shown two different types of expressing languages; finite automata and regular expressions. They, in fact, represent the same language class which is the class of regular languages. To show this formally, we need to give two lemmas that will also be used in the next section. First lemma states that every language defined by a finite automaton can also be defined by a regular expression. Second lemma states the inverse of the first lemma. That is, every language defined by a regular expression can also be defined by some finite automaton.

Lemma 1. If $L = L(A)$ for some finite automaton A, then there is a regular expression R such that $L = L(R)$.

Proof. Suppose that A's states are $\{1, 2, \ldots, n\}$ for some integer n. Let us use $R_{ij}^{(k)}$ as the name of the regular expression whose language is the set of strings w such that w is the label of a path from state i to j in A, and that path has no intermediate node whose number is greater than k. To construct the expressions $R_{ij}^{(k)}$, we use the following inductive definition, starting at $k = 0$ and finally reaching $k = n$.

Basis: The basis is $k = 0$. Notice that all states are numbered 1 or above. So, the restriction on the path is that the path must have no intermediate states. There are two types of path that meet such a condition:

1. An arc from node (state) i to node j.

2. A path of length 0 that consists of only some node i.

If $i \neq j$, then only first case is possible. We must examine the FA A and find those input symbols a such that there is a transition from state i to state j on symbol a.

1. If there is no such symbol a, then $R_{ij}^{(0)} = \emptyset$.

2. If there is exactly one such symbol a, then $R_{ij}^{(0)} = a$.

3. If there are symbols $a_1, a_2, \ldots, a_k$ that label arcs from state i to state j, then $R_{ij}^{(0)} = a_1 + a_2 + \cdots + a_k$.

If $i = j$, then the legal paths are the path of length 0 and all loops from i to itself. Such length is shown by the regular expression ϵ. Then, in case (1) the expression becomes ϵ, in case (2) the expression becomes $\epsilon + a$, in the third case the expression becomes $\epsilon + a_1 + a_2 + \cdots + a_k$.

Induction: Suppose there is a path from state i to j that goes through no state higher than k. There are two possibilities.

1. The path does not go through the state k. Then, the label of the path is in the language of $R_{ij}^{(k-1)}$.

2. The path goes through state k at least once. We can break the path into several pieces. Then, the set of labels for all paths of this type is represented by the regular expression $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$.

When the expressions for the paths of the two types above are combined, we have the expression

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

for the labels of all paths from state i to state j that go through no state higher than k. ■

We showed how to find the regular expression of any given FA. In fact, the method shown above may get complicated when the state size gets larger. To make the conversion more easily, we may use another method by eliminating the states of the FA. This method seems more practical than the first one. However, we do not show this method here. You may refer to [11] for FA to RE conversion by state elimination. Now, we give the second lemma which will be used together with the first lemma later on. We now show that every language L that is $L(R)$ for some regular expression R, is also $L(A)$ for some FA A. The proof is by structural induction on the expression R.

Lemma 2. Every language defined by a regular expression is also defined by a finite automaton.

Proof. Suppose $L = L(R)$ for a regular expression R. We show that $L = L(A)$ for some FA A with:

1. Exactly one final state.
2. No incomming arcs to the initial state.
3. No outgoing arcs from the final state.

The proof is by structural induction on R.

Basis: The basis has three parts. These cases are shown in Figure 2.1.

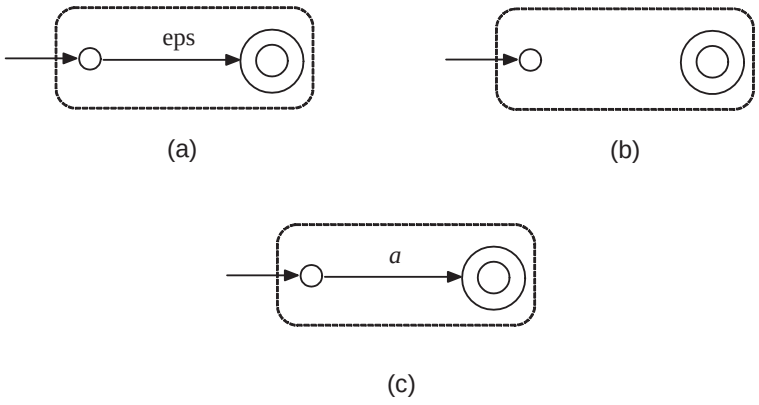

Figure 2.1: The basis of the construction of an automaton from a regular expression. Epsilon moves are indicated as 'eps'.

Part (a) in Figure 2.1 shows how to construct the automaton for the regular expression ϵ. It it clear that the language of the automaton is $\{\epsilon\}$. Part (b) shows how to construct the automaton for the regular expression $\emptyset$. Obviously, language of the automaton is $\emptyset$ in this case because there are no reachable paths from start state to the final state. Part (c) shows how to construct the automaton for the regular expression a. In this case, the language of the automaton consists of only the string a.

Induction: Similar to basis, inductive step has three parts is shown in Figure 2.2.

There are four cases in the inductive step.

- If the regular expression is $R + S$, we use part (a) in Figure 2.2 to construct it.

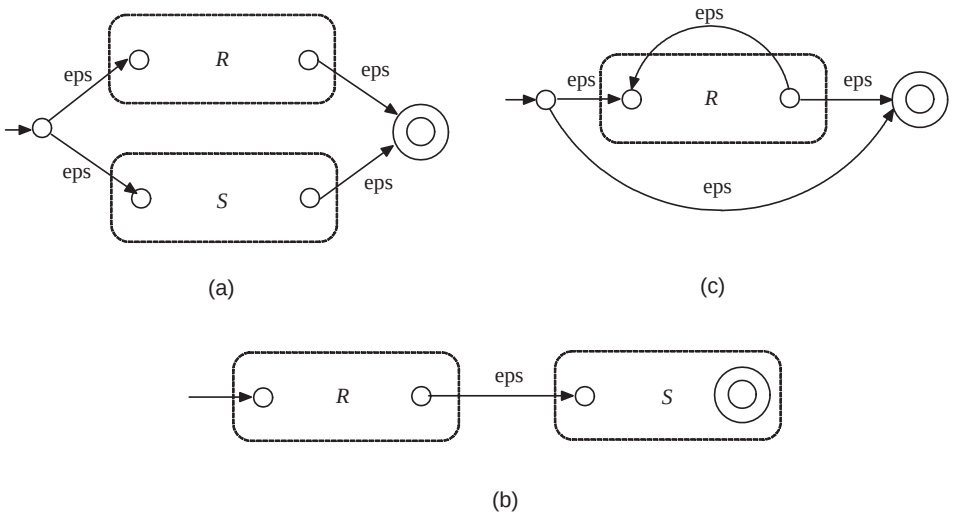

Figure 2.2: Induction to construct finite automaton from a regular expression.

- If the regular expression is RS, then we use part (b) to construct it.
- If the regular expression is R^*, then we use part (c) to construct it.
- If the regular expression is (R), then the automaton that defines R is used for (R) because the parentheses do not change the language.

At the end, the constructed automaton satisfies the inductive hypothesis. ■

2.3 Algebraic Properties of Regular Expressions

Regular expressions have a number of algebraic laws. In fact, they are quite similar to the laws of arithmetic if we think of the set-theoretic operations such as union and concatenation as addition and multiplication, respectively. A summary of algebraic laws for regular expressions is given below.

Associativity and Commutativity:

Commutativity: Let L_1 and L_2 be two regular expressions. Then,

$L_1 + L_2 = L_2 + L_1$.

Associativity for union: Let L_1, L_2 and L_3 be regular expressions. Then, $(L_1 + L_2) + L_3 = L_1 + (L_2 + L_3)$.

Associativity for concatenation: Let L_1, L_2 and L_3 be regular expressions. Then, $(L_1L_2)L_3 = L_1(L_2L_3)$.

Identity and Annihilator:

Let L be a regular expression. Then,

$\emptyset + L = L + \emptyset = L$ which asserts that $\emptyset$ is the identity element for union.

$\epsilon L = L\epsilon = L$ which asserts that ϵ is the identity element for concatenation.

$\emptyset L = L\emptyset = \emptyset$ which asserts that $\emptyset$ is the annihilator for concatenation.

Distributive Laws:

Let L_1, L_2 and L_3 be regular expressions. Then,

$L_1(L_2 + L_3) = L_1L_2 + L_1L_3$, and

$(L_3 + L_2)L_1 = L_3L_1 + L_2L_1$.

The Idempotent Law:

An operator is called idempotent if the result of applying it to two of the same values as arguments is that value. An assertion can be as follows.

Let L be a regular expression. Then, $L + L = L$ which asserts the idempotent law for union.

Closure Laws:

Let L be a regular expression. Then, $(L^*)^* = L^*$.

$\emptyset^* = \epsilon$.

$\epsilon^* = \epsilon$.

$L^+ = LL^* = L^*L$, where L^+ is defined as $L + LL + LLL + \cdots$.

$L^* = L^+ + \epsilon$.

So far regular languages, regular expressions, their model of computation counterpart called finite state automata, and properties of regular languages/expressions have been reviewed. In the next section, we describe the unary state change operator which manipulates the given finite automaton according to some specification. This transformation introduces a new set of languages called State Change Languages (SCL),

hence a new model of computation counterpart called State Automata (SA). The relationship between the regular set and SCL will also be shown.

GCRIS

CHAPTER 3

STATE CHANGE LANGUAGES

In this section, the formal definition of the state change operator (simply we call it the C operator) followed by an illustration is given. Then, the state change language class, its restrictions and properties are given. One of the main ideas of this section is to show that the obtained language by the transformation of the given finite automata using the C operator, is a proper subset of RL. We also discuss the alphabet change due to transformation of the given FA. Before we start to explain the state change operator, it is better to explain what a unary operator really means with some examples. Unary functions are mappings with one argument. If the elements of the domain are mapped to the elements in the same set (i.e. $U : A \rightarrow A$), then we call them *operators*. The term *function* is more general. Typical examples of unary functions are absolute values, logarithms, sin and cos functions. Their domain may not be a single argument, but map to tuples or other structures. For example, one may define a unary operator U which gets an integer and maps it to a pair consisting the number itself and its inverse. So, the domain set should be defined as $\{\ldots,-2,-1,0,1,2,\ldots\}$ which is equal to the set of integers, $\mathbb{Z}$. Then, the range set should be defined as $\{\ldots,(-2,2),(-1,1),(0,0),(1,-1),(2,-2)\ldots\}$. This is not limited with our example. The set can be extended to any level. As we will see later on, the state change operator acts on finite state machines. From the previous part, we defined finite state machines as 5-tuples. So, our domain consists of 5-tuples such as $\{(\Sigma_1, Q_1, \delta_1, q_{0_1}, F_1),(\Sigma_2, Q_2, \delta_2, q_{0_2}, F_2),\ldots,(\Sigma_n, Q_n, \delta_n, q_{0_n}, F_n)\}$. Still, the argument is considered to be single. Here, we see the importance of discrete structures in theoretical computer science.

3.1 State Change Operator and State Automata

Definition 4. A *State Automaton* (SA) $(Q, \Sigma', \delta', q_0, F)$ is a non-deterministic finite automaton with the following restrictions:

1. $\Sigma' = \{0, 1\}$ is a restricted two symbol alphabet,
2. Transition function can include transition rules either in the form
   $\delta'(p, 0) = \{p\}$, where $p \in Q$ or,
   in $\delta'(p, 1) = R$, where $R \in P(Q - \{p\})$, or both.

Now we will define the C operator, which acts on the transitions of NFA and produces another automaton. The input to the operator is a finite automaton, and the output is another automaton, but these two automata are related. The C operator is defined as follows.

Definition 5. *State change operator*, C, is a surjective mapping $C : NFA \rightarrow SA$ such that, for a given $A = (Q_A, \Sigma_A, \delta_A, q_{0_A}, F_A) \in NFA$, it produces $C(A) = B = (Q_B, \Sigma_B, \delta_B, q_{0_B}, F_B) \in SA$ such that, $Q_B = Q_A$, $\Sigma_B = \{0, 1\}$, $q_{0_B} = q_{0_A}$, $F_B = F_A$, and for each $\delta_A(p, a) = R$,we describe δ_B as;

$$\begin{aligned} &\text{if } p \notin R, \text{ then} && \delta_B(p, 1) = R \\ &\text{if } p \in R, \text{ then} && \delta_B(p, 0) = \{p\},\ \delta_B(p, 1) = R - \{p\} \end{aligned}$$

being new state transition rules.

Simply, the C operator modifies the transition function of a given NFA by mapping its input symbols to $\{0, 1\}$. The transformation takes $O(|\Sigma||Q|^2)$ steps. To illustrate the transformation, an example is given below.

Example 1. Let $A = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$ be an NFA where, the transition function δ is defined by,

$\delta(q_0, a) = \{q_0\}$

$\delta(q_0, b) = \{q_1\}$

$\delta(q_1, a) = \{q_0\}$

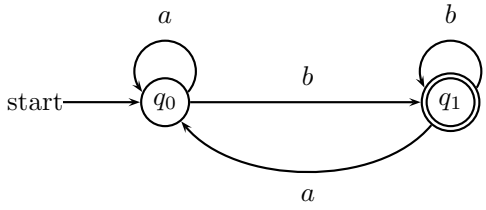

Figure 3.1: A finite automaton A.

$\delta(q_1, b) = \{q_1\}$
The transition diagram representation of A is shown in Figure 3.1.

Notice that in A state change occurs either when the current state is q_0 and the input is b, or when the current state is q_1 and the input is a. Then, by applying C operator, the given NFA is transformed by relabeling its input symbols. The transition diagram representation of new automaton is given in Figure 3.2.

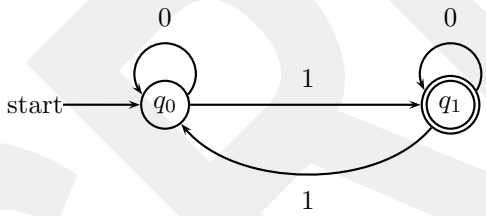

Figure 3.2: State automaton $B = C(A)$.

During the transformation, the number of states and the number of transitions are preserved. Yet, we may have transitions that duplicate, so we simply eliminate one of them. The output automaton, say B, has a fixed alphabet $\Sigma = \{0, 1\}$, which in fact may be different than $\{0, 1\}$. It could be any two symbols that represent two discrete events, i.e. state change and no state change. Therefore, one could give $\{a, b\}$, or $\{\Delta, \square\}$, or $\{\lambda, \partial\}$ or any set S, $|S| = 2$, that is capable of representing two discrete events. Finally, its number of states remains the same as A. Obviously, $L(A) \neq L(B)$ since the automata take their description from different domains. $L(A) = \{w|\ w$ ends with b$\}$, while $L(B) = \{w|\ w$ has odd number of 1's$\}$. If we try to write their regular expression, it will be $R_A = (a^* + bb^*a)^*bb^*$ and $R_B = (0^* + 10^*1)^*10^*$. The state automaton describes the state change characteristic that forms a basis for power consumption description of NFA under consideration. Regarding the algebraic properties, it is easy to see that

C is idempotent, i.e. $C(C(A)) = C(A)$ for any NFA A. It is also clear to see that C has a many-to-one functional property. Therefore, C is not invertible. That is, since C is surjective we cannot obtain the input from the output. It would be possible if it were injective. For the following discussion, we give the following definition.

Definition 6. Let A and B be two alphabets. A function $f : A^* \rightarrow B^*$ is called a *language morphism* over languages $L(A)$ and $L(B)$ such that $f(\epsilon) = \epsilon$, and $f(w) = f(u)f(v)$ for all $w, u, v \in A^*$ such that $w = uv$ satisfying $w \in L(A)$ and $f(w) \in L(B)$.

The C operator may not always provide a morphism of languages.

Proposition 1. Let $A = (Q_A, \Sigma_A, \delta_A, q_{0_A}, F_A) \in NFA$ without ϵ transition such that $L(A) \neq \emptyset$. There exists a morphism $h : \Sigma^* \rightarrow \{0, 1\}^*$ defined over $L(A)$ and $L(C(A))$, respectively if and only if for each $a \in \Sigma_A$, a transition rule $\delta_A(p, a) = R$ should satisfy either $R = \{p\}$ or $R \cap \{p\} = \emptyset$ for all $p \in Q_A$.

To make the latter statement clear, consider Example 1. We show that no such morphism g exists such that if $w \in L(A)$, then so is $g(w) = g(x)g(y) \in L(C(A)) = L(B)$, where $w, x, y \in \Sigma^*$, and $w = xy$. As a counter example, consider the case $w = bb \in L(A)$. If there exists a morphism g, then it should satisfy the conditions that are specified, i.e. $g(bb) = g(b)g(b)$ must be in $L(B)$. However, bb gets the value 10, in this case, in automaton B. This means that we cannot assign a unique value for the symbol b.

Introducing the C operator reveals the fact that there exists at least one interpretation related with the syntax of finite automata, and that is the power consumption. It is worth noting that one can apply the C operator to any model of computation, through some extensions. The problem is that the definition of the operator must be extended so that it precisely defines necessary parameters for each computational model. One can see that the current definition of the C operator is not complete over Turing machines. Besides the state changes of its finite control unit, Turing machine has other mechanics like writing on/reading from tape or moving tape head that may form basis for alternative power consumption characteristics. It appears that the C operator is universal for standard finite automata. That is, it is a universal operator to investigate its abstract power consumption, because there is no other option than its state

changes which is the only implication for its power consumption. We can also say that different computations, and models of computation as well, consume different level of power consumption. In the following section, a new class of languages, SCL, obtained by the surjective mapping, is introduced.

3.2 State Change Languages and Expressions

Any state expression, defining a state automaton, is also a regular expression, but not vice versa. For example, a state expression cannot include 0^k expression (i.e. $0, 00, 000, \ldots, 00\ldots0$) for any constant $k > 0$ in it, by definition. In other words, we have defined a 0 transition as "no state change" which means that in any state q, the automaton should change its state to q again which defines an infinite loop. So, we cannot define infinite loop with a finite number of transitions. Whenever the regular expression of the automaton is written, it will always be as 0^* instead of 0^k.

Definition 7. A language is in *SCL* if some state automaton recognizes it.

Theorem 1. The set of state change languages is a proper subset of regular languages. That is, $SCL \subsetneq RL$.

Proof. First we have to show that every language in SCL is also in RL. This is obvious since every state automaton is an NFA. Next, we have to show that there exists at least one regular expression E such that $L(E) \neq L(C(A_i))$ for all $A_i \in NFA$. That is, we have to show that there exists at least one regular language $L(E)$ that cannot be recognized by any state automaton. Consider the language $L(0) = \{0\}$. It means that some state automaton B must accept the input 0, and reject any other given input. In other words, $L(B) = \{0\}$ should hold. This contradicts with the fact that B cannot hold any transition rule such as $\delta(p, 0) = R - \{p\}$ which is necessary for the generation of $\{0\}$. ■

Corollary 1. If there exists a state automaton A, then there exists a state expression E, such that $L(A) = L(E)$.

Proof. The proof is trivial. We must show that there exists a SE E if there exist a SA A satisfying $L(A) = L(E)$. Since every state change language is a regular language by Theorem 1, we can use Lemma 1 to convert A to E. ■

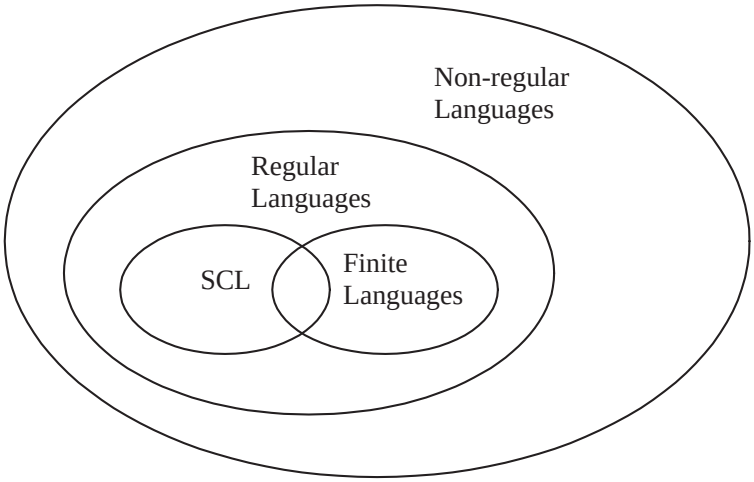

Figure 3.3: Relationship among language classes involving SCL.

As shown in Figure 3.3, SCL can be classified as a subset of regular languages. Note that SCL is an infinite set. However, not all finite languages are in SCL. For example, the set {0} is in the finite language set, but not in SCL. On the other hand, $L(0^*)$ is in SCL, but not in the finite language set. $L(01^*)$ is neither in SCL nor in finite languages. Conversely, {1} is both in SCL and finite languages.

GCRIS

CHAPTER 4

CLOSURE PROPERTIES

This section considers the closure properties of SCL. It is known that regular languages are closed under some operations including union, intersection, complement, difference, reversal, kleene star, concatenation, homomorphism, and inverse homomorphism [11]. Recall that a *homomorphism* is a map $h : \Sigma^* \to \Gamma^*$ for two alphabets Σ, Γ, such that $h(w) = h(a_1)h(a_2)\cdots h(a_n)$ for all $w = a_1a_2\cdots a_n \in \Sigma^*$, where $a_i \in \Sigma$, $1 \leq i \leq n$. The set of SCL is not closed under homomorpism and inverse homomorphism. In fact, the alphabet change in the definition is the point where the problem arises. If we let $\Sigma = \{0, 1\}$, and $\Gamma = \{x, y, z\}$, then we have a problem with representing state changes. Remember that we only have two symbols in SCL. The third symbol is undefined in SCL, and this results in any such string, containing three or more different symbols, fail to be interpreted since the operator acts on state changes, i.e either we have a state change or no state change. Similarly, SCL is not closed under any inverse homomorphism $h : \Gamma^* \to \Sigma^*$, for every Γ such that $|\Gamma| \geq 3$.

Theorem 2. If L and M are SCL, then so is $L \cup M$.

Proof. The proof is by product construction of two state automata. If L and M are in SCL, then there exist state automata $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ that recognize them respectively. For $L \cup M$, we construct an automaton A that simulates A_1 and A_2. The states of A are of the form (Q_x, Q_y) such that $Q_x \in P(Q_1)$ and $Q_y \in P(Q_2)$. Furthermore, δ of A is defined as $\delta : Q_1 \times Q_2 \times \Sigma \to P(Q_1) \times P(Q_2)$. The start state of A is the pair (q_1, q_2), i.e. the start states of A_1 and A_2. A should accept if and only if either of the automata accepts. Therefore, the final states of A consist of pairs (Q_{xf}, Q_{yf}) such that either $Q_{xf} \in P(Q_1) \cap F_1 \neq \emptyset$ or $Q_{yf} \in P(Q_2) \cap F_2 \neq \emptyset$, or both. So, we define $A = (Q_x \times Q_y, \Sigma, \delta_A, (q_1, q_2), Q_{xf} \times Q_{yf})$ and $\delta((p, q), a) =$

$(\delta_1(p,a),\delta_2(q,a))$. Thus, by this way, A accepts $L \cup M$. ■

Let us now consider closure by intersection. It is similar to closure by union, but with a little modification.

Theorem 3. If L and M are in SCL, then $L \cap M$ is in SCL.

Proof. Similar to closure by union, we again use product construction here. If L and M are in SCL, then there exist state automata $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ that recognize them respectively. We construct an automaton A that simulates A_1 and A_2 such that A recognizes $L \cap M$. The states of A are of the form (Q_x, Q_y) such that $Q_x \in P(Q_1)$ and $Q_y \in P(Q_2)$. We define the transition function δ of A as $\delta : Q_1 \times Q_2 \times \Sigma \to P(Q_1) \times P(Q_2)$. The start state of A is the pair (q_1, q_2). Here, A should accept if and only if both of the automata accept. So, the final states of A consist of pairs (Q_{xf}, Q_{yf}) such that both $Q_{xf} \in P(Q_1) \cap F_1 \neq \emptyset$ and $Q_{yf} \in P(Q_2) \cap F_2 \neq \emptyset$ holds. So, we define $A = (Q_x \times Q_y, \Sigma, \delta_A, (q_1, q_2), Q_{xf} \times Q_{yf})$ and $\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a))$. Hence, A accepts $L \cap M$. ■

State automata are non-deterministic finite automata, by definition. However, depending on the input automaton, the state automaton can also be deterministic. Interestingly, deterministic state automata (DSA) is less powerful than its nondeterministic counterpart. Many languages that are recognized by non-deterministic state automata, cannot be recognized by deterministic state automata. The problem is based on the following definition.

Definition 8. A finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ is said to be *incomplete* if $\delta(q,a) = \emptyset$ for some $q \in Q$ and $a \in \Sigma$. It is *complete*, otherwise.

If an automaton is deterministic, it must be complete. If the automaton is incomplete, then it is nondeterministic. However, being that an automaton is complete does not imply that it is deterministic. So, we give the following theorem.

Theorem 4. There are state change languages that cannot be recognized by any deterministic state automata.

Proof. A simple counter example is the language $\{1\}$. Although it can be recognized by some nondeterministic state automaton (NSA) whose δ consists of a single transition $\delta(q_0, 1) = \{q_f\}$, where q_0 is the start state and q_f is the final state, this automaton

is incomplete. Hence, it is obviously nondeterministic. If we try to make this automaton complete, then all states should self-loop with a 0 transition which directly changes the language. ■

We conclude the following from the above theorem.

Corollary 2. There is no conversion from NSA to DSA for all state change languages.

It is worth noting that eliminating transitions with empty string, does restrict constructions for proving closure properties. It arises a philosophical question that is whether the empty string in formal language theory really helps us to construct unique mathematical objects, finite automata in particular, to preserve closure properties. Allowing empty string transitions in state automata makes things easier and brings more closure properties. That is, if we leave each epsilon transition as it is, hence add the rule $\delta_B(p, \epsilon) = R$ if $(a = \epsilon)$ to the definition of C operator, obvious constructions lead us to have more closure properties.

Another fact about the set of state change languages is that it is not closed under complementation. Let us again consider the language $L = \{1\}$. Then, for instance, it must be the case that $010 \in \overline{L}$. But the only possible state automaton to recognize this string must also recognize the string 1 which contradicts with the fact that $1 \notin \overline{L}$. Therefore, we say that SCL is not closed under complementation.

Unlike RL, SCL is not closed under complementation, homomorphism and inverse homomorphism. We do not currently know whether or not they are closed under other operations. Next, we show how closure properties change when we allow empty string in state automata.

4.1 State Automata with ϵ-Transitions

By definition, state automata do not include transitions with empty string. Allowing empty string in state automata would contradict the intuitive notion of abstract power consumption since state changes may still occur with ϵ-transitions in any given finite automata. That is, internal state changes caused by ϵ-transitions in a finite automaton

should also be considered which is already handled by the C operator. So, the C operator treats empty string like other input symbols. On the other hand, ϵ-transitions are independent from the input string which does not violate the notion of abstract power consumption of a finite automaton for a given particular input string. This requires us to use ϵ-transitions explicitly in state automata. In this case, we leave each ϵ transition in a given finite automaton as it is. Therefore, the definition of state change operator should be extended. We give the following definition for a new C with ϵ moves.

Definition 9. *State change operator with ϵ moves, C_ϵ,* is a surjective mapping $C_\epsilon : NFA \rightarrow SA$ such that, for a given $A = (Q_A, \Sigma_A, \delta_A, q_{0_A}, F_A) \in NFA$, it produces $C(A) = B = (Q_B, \Sigma_B, \delta_B, q_{0_B}, F_B) \in SA$ such that, $Q_B = Q_A$, $\Sigma_B = \{0, 1\}$, $q_{0_B} = q_{0_A}$, $F_B = F_A$, and for each $\delta_A(p, a) = R$,

$$\begin{aligned} \delta_B(p, \epsilon) &= R && \text{if } (a = \epsilon) \\ \delta_B(p, 1) &= R && \text{if } (a \neq \epsilon) \wedge (p \notin R) \\ \delta_B(p, 0) = \{p\},\ \delta_B(p, 1) &= R - \{p\} && \text{if } (a \neq \epsilon) \wedge (p \in R) \end{aligned}$$

Let us call the set of languages recognized by state automata with ϵ-transitions, ϵ-SCL. Clearly, every language in SCL is also in ϵ-SCL. Obviously, using ϵ in state automata gives modularity in proving the closure properties. Many of them become trivial to construct. Union and intersection proofs are also valid in this case. Other than these two, we give the following theorems for closure properties of state automata with ϵ transitions.

Theorem 5. If L and M are in ϵ-SCL, then so is $L \circ M$.

Proof. Let $S_L = (Q_l, \{0, 1\}, \delta_l, q_{ls}, F_l)$ and $S_M = (Q_m, \{0, 1\}, \delta_m, q_{ms}, F_m)$ be the state automaton of L and M respectively. The construction for the concatenation of regular languages applies here. We construct an automaton $S_N = (Q_l \cup Q_m, \{0, 1\}, \delta_n, q_{ls}, F_m)$, where $\{\delta_l\} \cup \{\delta_m\} \subseteq \{\delta_n\}$, with an additional transitions $\delta_n(q_{lf}, \epsilon) = \{q_{ms}\}$ for each $q_{lf} \in F_l$.

As shown in Figure 4.1, similar to the concatenation construction for regular languages, the resulting automaton is a state automaton with ϵ-transition. The input string will be accepted if and only if it has an accepting computation on the automaton S_N which is $L \circ M$. ■

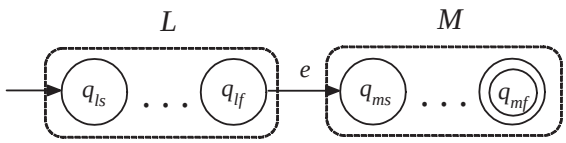

Figure 4.1: State automaton S_N recognizing the language $L \circ M$.

Let $w = a_1 a_2 \dots a_n$ be a string over Σ. Then, its reversal is $w^R = a_n a_{n-1} \dots a_1$. For example if $w = 00110$, then $w^R = 01100$. We say that the reversal of a language L is $L^R = \left\{ w^R | \, w \in L \right\}$.

Theorem 6. If L is in ϵ-SCL over Σ, then L^R is also in ϵ-SCL.

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a state automaton, such that $L(A) = L$. We define the reversal of A as $A^R = (Q, \Sigma, \delta^R, q_0, F)$. Since $\{p\} \in \delta^R(q, a) \Leftrightarrow \{q\} \in \delta(p, a)$, we have $L(A^R) = (L(A))^R = L^R$. ■

Theorem 7. If L is in ϵ-SCL, then so is L^*.

Proof. Let $S = (Q, \Sigma, \delta, q_0, F)$ be an arbitrarily given state automaton, where $F = \{q_{f_1}, \dots, q_{f_i}\}$ for some $1 \leq i \in \mathbb{N}$. We construct another state automaton $S' = (Q', \Sigma, \delta', q'_s, F')$, given in Figure 4.2, where $Q' = Q \cup \{q'_s\}$, and $F' = \{q'_s\}$, we add the following transitions to obtain δ' so that $\{\delta\} \subsetneq \{\delta'\}$ is satisfied:

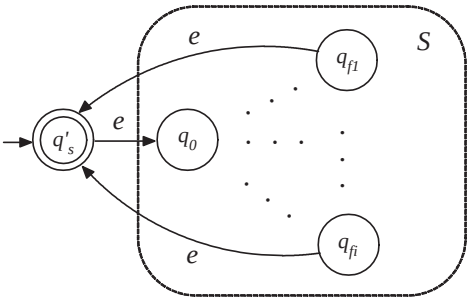

Figure 4.2: State automaton S' recognizing the language $(L(S))^*$.

$\delta'(q'_s, \epsilon) = \{q_0\}$, and $\delta'(q_f, \epsilon) = \{q'_s\}$ for every $q_f \in F$. The resulting automaton is clearly a state automaton with additional ϵ-transitions. ■

We see that constructions become trivial with ϵ-transitions. We do not know whether or not SCL is closed under concatenation and reversal. However, it is again worth

noting that regardless of whether the automaton is with or without ϵ moves, deterministic or nondeterministic, the set of state change languages are not closed under complementation and string homomorphism.

GCRIS

CHAPTER 5

DECIDABILITY OF STATE CHANGE LANGUAGES

Decidability plays an important role in theory of computability and mathematical logic in order to define the limitations of mathematical theories. It is almost a tradition to discuss the decidability of a proposed formal language set, when given one. In this section we consider two decision problems regarding SCL. One of them is deciding the language emptiness, other one is the problem of deciding whether or not the language of a given Turing machine is a state change language. We will see that one of them is decidable, where the other remains to be undecidable. Before showing these problems, let us first discuss the notion of uncomputability.

Undecidability, the opposite extreme of decidability, is also an important concept in theory of computation and logic. We have the notion called countability in the very heart of computability. We say that a set A is countable if it is finite or there exist a bijection $f : A \rightarrow \mathbb{N}$. Every countable set is enumerable. That is we can assign a number $n \in \mathbb{N}$ for each element of the enumerable set. The idea of uncomputability goes back to the work of Georg Cantor on Set Theory when he proved the existence of uncountable sets in the late 19th century with a powerful method called *diagonalization* [32]. Obviously, at that time the term computation was not known properly. In his proof, he considered an infinite sequence of the form $(x_1, x_2, \ldots)$, where each $x_i = \{0, 1\}$.

$$S_1 = \mathbf{1}, 1, 0, 0, 1, 0, 1, 1, \ldots$$
$$S_2 = 0, \mathbf{0}, 1, 0, 1, 1, 0, 0, \ldots$$
$$S_3 = 0, 0, \mathbf{0}, 1, 0, 1, 0, 0, \ldots$$
$$S_4 = 1, 0, 0, \mathbf{0}, 1, 1, 1, 1, \ldots$$

$$
\begin{aligned}
S_5 &= 0, 1, 1, 0, \mathbf{1}, 0, 1, 1, \ldots \\
S_6 &= 1, 1, 0, 1, 0, \mathbf{0}, 0, 1, \ldots \\
S_7 &= 1, 1, 0, 1, 0, 1, \mathbf{0}, 0, \ldots \\
S_8 &= 1, 0, 1, 0, 1, 0, 0, \mathbf{1}, \ldots \\
\vdots &\qquad\quad \vdots \qquad\quad \ddots \\
S_d &= 0, 1, 1, 1, 0, 1, 1, 0, \ldots
\end{aligned}
$$

We construct S_d by taking the complement of the diagonal values. However a contradiction occurs at the position where diagonal value meets the value in S_d. Whatever the symbol is, it should be the opposite of itself in either case, it is a contradiction. It is like saying "This statement is false". Therefore, one can never present a complete list of binary sequences.

In 1931, Kurt Gödel [12] published his famous Incompleteness Theorems with a breakthrough paper, On Formally Undecidable Propositions of Principia Mathematica and Related Systems, which became one of the most celebrated results in mathematics, logic, and of course theoretical computer science. Gödel showed that any formal axiomatic system that is capable to represent arithmetics (such as the Peano Arithmetic and Zermelo-Frankel), is either inconsistent or incomplete. In other words, there will always be statements, say G such that neither G nor $\neg G$ can be proved within the system no matter how large the set of axioms is. Because if we add the undecidable statement G to the system, say S, as an axiom, we obtain a new system, say S'. So, this stronger system again has the same sickness. In fact, Gödel used the idea of liar paradox, and diagonalization which was introduced by Cantor. He assigned a specific number, called Gödel number, to each formula and proof. Gödel showed the unachievability in terms of provability in formal axiomatic systems. Alan Turing, then introduced Turing machines which are assumed to have equivalent power to recursive functions. In a point of view of theoretical computer science, Gödel's result says that there are undecidable problems which cannot be solved by any reasonable model of computation. As a fundamental reason, there are more languages than Turing machines. So, the timeline of undecidability starts with Cantor who introduced infinite sets and uncountability. Then, Gödel showed the unprovability of mathematical statements. With Turing machines we define uncomputability by the problem

known as the halting problem.

Now, after this introduction, we show two decision problems related to SCL. One of them is decidable where the other remains to be undecidable. The membership problem is also decidable for SCL although a particular section it not dedicated for it.

5.1 Emptiness Checking

This part focuses on the decision problem called the emptiness checking of state languages. In fact, the problem is deciding whether or not $L = \emptyset$. The answer is obvious. $\emptyset$ is empty and all other state languages are not. If we are given a state automaton, we can think of the problem as graph reachability. Simply, if a final state is reachable from the start state then we say that the language is not empty, and it is empty otherwise.

Proposition 2. For any given SA S, determining whether or not $L(S) = \emptyset$ is decidable.

Proof. First, let us define the reachability recursively as follows:

Basis: The start state is reachable from the start state.

Induction: If a state q is reachable from the start state and there exist a transition from q to p, then p is reachable.

Complexity of state reachability for a given state automaton is $O(n^2)$ for n states. If we are given a state expression then we have to convert the expression to its equivalent state automaton, then apply the process given above. Though, we can apply the following recursive rule to check if the given state expression denotes empty language.

Basis: $\emptyset$ is the empty language. Nothing else is empty language.

Induction: Suppose S is a state expression.

1. If $S = R + M$, then $L(S)$ is empty iff both $L(R)$ and $L(M)$ are empty.
2. If $S = RM$, then $L(S)$ is empty iff either $L(R)$ or $L(M)$ is empty.
3. If $S = R^*$, then $L(S)$ is not empty because kleene star includes the empty string ϵ.
4. If $S = (R)$, then $L(S)$ is empty iff $L(R)$ is empty. ■

It is again worth noting that the set of SCL is decidable and since $SCL \subsetneq RL$. That is, given a string w and a state automaton A, determining whether or not $w \in L(A)$ is decidable. Furthermore, it is also decidable that given any expression in any form whether or not it is a state expression. By constructing a pushdown automata, we can decide this problem.

5.2 Language of the Turing machine

In this part, we consider the problem of deciding whether or not the language of the given Turing machine is a state change language. We show that this problem is undecidable. Let us call this problem S for the time being. We will use the notion of reducibility here. We will reduce an undecidable problem H to S. By reduction, we will have the fact that the solution to the problem S implies the solution to the problem H. Since there is no solution to H, we deduce the fact that S must also be undecidable. Proving undecidability always involve the following logical argument.

1	$S \rightarrow H$	premise
2	S	assumption
3	H	MP; 1,2
4	$\neg H$	premise
5	$\perp$	
6	$\neg S$	RAA; 2-5

First, let us give the definition of reducibility which plays an important role in theory of computation used in classifying the decidability and computational complexity of problems.

Definition 10. Let A and B be two languages. We say that A is *reducible* to B, denoted as $A \leq B$, if there is a computable function[1] $f : \Sigma^* \rightarrow \Sigma^*$, such that $w \in A \iff f(w) \in B$ for all w.

Let L_1 and L_2 be two languages. If $L_1 \leq L_2$, then L_2 is at least hard as L_1. The direction of reducibility is important. We usually reduce from specific to general which allows

[1] A function $f : \Sigma^* \rightarrow \Sigma^*$ is said to be *computable* if it halts on a Turing machine for every input $w \in \Sigma^*$.

us to infer that if L_1, which is a subset of L_2, is undecidable, then we guarantee that so is L_2.

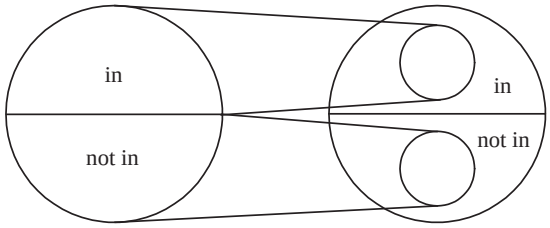

Figure 5.1: Reduction from L_1 to L_2. The solution of L_2 is the solution of L_1.

5.3 An Undecidable Problem

In this part, we prove the fact that given a Turing machine M, determining whether or not the language of M is a state change language, is undecidable. One may wish to know whether or not a given Turing machine is equivalent to some state automaton. This problem can be regarded as a type of minimization problem. Given a Turing machine M, determine whether or not it can be minimized to some state automaton S, such that $L(M) = L(S)$. In other words, the same given language can be recognized by some state automaton instead of by a Turing machine, to ensure that the computation would take less theoretic power for all instances, if computational model complexity is proportional to abstract power consumption. This problem is no different than the problem of determining whether or not the language of a given Turing machine is a state change language. There is a theorem [22] which says that there is no solution to this problem.

Definition 11. A language is *recursively enumerable* if it can be recognized by some Turing machine.

Theorem 8 (Rice's Theorem)**.** Every nontrivial property of the recursively enumerable languages is undecidable.

The undecidability of language related problems regarding Turing machines, in fact, has shown to be common. In Rice's theorem, nontrivial properties refer to the questions related to language (or some other functional property) of the Turing machine.

Even the problem of determining whether or not the language of the given Turing machine is finite, has been shown to be undecidable. So, consider the problem of determining given a Turing machine, whether the language of the Turing machine is in SCL or not. Let us call this problem S_{TM}, and represent the string encoding of Turing machines with the notation $\langle M \rangle$. More formally, the following set is undecidable.

$$S_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is in SCL}\}.$$

Before proving the undecidability of S_{TM}, let us first prove the undecidability of the following problem. The problem is called the halting problem on empty tape. Let us call this problem B_{TM} which is defined as

$$B_{TM} = \{\langle M \rangle \mid M \text{ is a TM and halts on } \epsilon\}.$$

Lemma 3. B_{TM} is undecidable.

Proof. We reduce the halting problem to B_{TM}. Let M be the Turing machine which decides whether or not it halts on the given input w. We construct a Turing machine M_w which writes w on its tape when it is started on the empty tape, then simulates M. If the computation halts, then *accept*, ttherwise, *reject*. Note that M_w halts on $w \iff M_w$ halts when started on empty tape. ■

So far, we have shown how to prove problems to be undecidable by reductions. Now, we prove the problem S_{TM} to be undecidable by using a similar reduction.

Theorem 9. S_{TM} is undecidable.

Proof. The proof is by reduction from B_{TM}. Given $\langle M \rangle$, we construct another Turing machine M', such that it simulates M on ϵ. If the computation halts, then the universal Turing machine runs the Turing machine M' on any string $w \in N$, where N is some non-state change language. Notice that M' halts either on w or on no strings, depending on whether M halts on ϵ or not. But there is no Turing machine that decides whether M halts on ϵ. This implies that there is no Turing machine that decides whether $L(M) = \emptyset$ which is a state change language, or N which is a non-state change language. ■

Obviously, this is a consequence of Rice's theorem. The problem may not look important by itself, but once it is related with the notion of abstract power consumption it becomes more interesting. The next section discusses abstract power consumption, uncomputable numbers, and a related undecidable problem.

GCRIS

CHAPTER 6

ABSTRACT POWER CONSUMPTION

Abstract mathematical models of computation have different resource consumptions. Two important and known resources are space and time. This concise chapter concerns with another important resource, *energy*.

6.1 Abstract Power and Complexity

In this concise section, the notion of abstract power consumption for finite automata is discussed. It has been mentioned that the C operator maps the given finite automaton to its state automaton, i.e. to its state change description. In fact, we consider state changes as a cost in terms of abstract power on finite automata. For this purpose we give the following.

Definition 12. A finite automaton F consumes a *unit abstract power* during its computation $\Delta = q_0a_1a_2\cdots a_n \vdash q_ka_2a_3\cdots a_n \vdash \cdots \vdash q_f\epsilon$, where $q_k \in Q$, $q_f \in F$, and $a_1, a_2, \ldots, a_n \in \Sigma$, if $q_k \neq q_{k-1}$ (i.e. the state at the computational step n differs from that at step $n-1$). *Total abstract power consumption of F* is the number of state changes in a run, and given by the expression $\sum_{i=1}^{n} dif(q_i, q_{i-1})$, where $dif(q_i, q_{i-1})$ is 1 if the states are different at ith computational step from the $i-1th$ computational step. $dif(q_i, q_{i-1})$ is 0, otherwise.

Finding the minimum total abstract power consumption of a given state automaton is nothing but to find shortest run (or path) from q_0 to any of the final states $q_f \in F$, that can be computed in polynomial time. Clearly, Definition 11 is not limited to finite automata. One can extend the operator for Turing machine. An example work is done

by authors in [23], in which a Turing machine variant is constructed to model energy consumption at algorithmic level in wireless systems. A more hardware treatment of energy complexity is given in [24]. The critical thing is that the operator or variant should be universal. In general, for any reasonable model of computation, including quantum Turing machine, deciding on a lower bound for their total abstract power consumption would be interesting in their own right.

We may define a new complexity notion, *power complexity*. The power complexity of finite automata is certainly bounded above by its time complexity. We may have three fundamental complexity notions then: time, space, and power. The three notions consider, respectively, how much time does the machine take to compute something, how much memory does it require to perform the computation, and how much energy, in principle, does it require to perform the computation. In the next section, we give a problem which is strongly undecidable in its own right.

6.2 An Uncomputable Number

Now, we show that there is a particular mathematical object, which is a real number related with a problem about abstract power consumption, happens to be undecidable. First, let us give the following definition about computable numbers which goes back to the work of Alan Turing [21].

Definition 13. A real number is *computable* if, given an integer n, there exists a computable function that outputs the first n digits of its decimal expansion.

Some real numbers can be computable. For example Pi is computable. The way it is computeed is that by just calculating it. Our result, in fact, is a consequence of Rice's theorem, and Gregory Chaitin's [13] work of irreducible structures in algorithmic information theory [25][26]. Chaitin, inspired from Gödel's incompleteness theorems, showed that there are mathematical truths which are true for no reason, i.e. has no structure or irreducible. Usually, mathematical structures can be reduced to a more compact form which are deducible from it, such as a set of axioms in a formal system. As Leibniz said, if something is true it is true for a reason which we call it as the proof of the fact. However, Chaitin, as well as Gödel, showed that there are constructible,

but uncomputable information that has no structure. In fact, what Chaitin meant was the source of undecidability is the randomness in mathematics, as well as the lack of structure. For example, irreducible information may look like an independent tosses of a fair coin. Consider a binary string of length n. If the string cannot be represented simpler than itself, then the program to compute the bits of that string should have n bits of axioms. There is a particular real number called the Chaitin's Omega Number, or simply Ω. Chaitin considered Turing's halting problem, then he constructed Ω, also called the halting probability, which represents the probability that a randomly chosen program will halt. He showed that Ω is uncomputable. That is, there is no algorithm that enumerates its digits. The Ω number relies on the existence of prefix-free universal computable functions [27]. Let P_F be the domain of a prefix-free universal computable function F. The omega number is defined as,

$$\Omega_F = \sum_{p \in P_F} 2^{-|p|}$$

Now consider the problem of determining whether or not the language of a given Turing machine is a regular language or not. If the language of the Turing machine was really some regular language, then it would be possible to construct a finite automaton accepting the same language. Let us think about the consequences.

Definition 14. A computational model M is said to be *abstract power minimizable* if $L(M') = L(M)$ by some M' such that M' consumes less abstract power than M for all inputs.

The above definition may sound a bit far fetched, but it is crucial to understand the following problem. We may say that descriptional complexity [33] of a model of computation is nothing but its own mechanical complexity. It is reasonable to say that Turing machine is more complex than a finite state machine. It seems that recognizing a larger set of languages requires more complex mechanisms. If descriptional complexity and the abstract power consumption are proportional, then this problem can be seen as abstract power minimization. This indirectly means that the same functionality can be done with less abstract power. Instead of Turing machine, we can recognize the same language by some finite automaton which is something that consumes less abstract power and has less descriptional complexity than any Turing

machine, by hypothesis. In fact this problem is strictly undecidable.

We can almost construct another instance of Ω related with the fact about Turing machines that are abstract power minimizable or not. Think about an infinite binary sequence where each bit represents the i^{th} Turing machine in size order.[1] We construct our real number as follows: If the i^{th} Turing machine is abstract power minimizable, then we put 1. Otherwise, we put 0. This number is uncomputable. Given some integer *n*, there exists no finite procedure that enumerates its digits. The number cannot be compressed smaller than it is. For *n* bits, we need *n* bits of axioms. This is another Ω number that we can almost construct in some way, but uncomputable. So, if we believe that descriptional complexity is proportional with abstract power consumption, then our problem is undecidable.

[1] The set of all Turing machines is countably infinite, we can obviously enumerate them.

CHAPTER 7

AXIOMATIZATION OF ABSTRACT POWER CONSUMPTION

At the beginning of our discussion, we said that formal languages such as regular languages or its subclasses were widely used in model checking. For example the authors in [17] presented an encoding for bounded model checking for regular properties as specification model. Using ω-regular languages which are recognized by Büchi Automata is the most common case in model checking of real time systems, especially with specifications represented in temporal logic formulas which is where timed automata [18] are considered. We will not consider such case in this work. Rather, we consider satisfiability on state automata, i.e. those always halt. Also we do not focus on Büchi automata since nondeterministic/deterministic conversions may lead to problems.

In this section, we give the logical description of ϵ-SCL generally. We also give an axiomatization of abstract power consumption over finite automata. Then, we use this formalism to ensure whether the given abstract power consumption property of a finite automaton satisfies the formal specification, represented in first-order predicate logic, which is a property of another finite automaton that we are looking an equivalent model for with respect to the specification.

7.1 Logical Description of SCL

Formal languages which are nothing but sets, can be described by models of computation, algebraic structures, and logical formulas [28][29][31]. Furthermore, all descriptions define the same set of languages. That is, no descriptional method is

more or less powerful than other in most cases. Here, we give a general description of state change languages with ϵ-transitions, shortly $\epsilon - SCL$, in terms of first-order logic. The logical description of SCL without ϵ-transitions is far more complex since we do not know whether it is closed under concatenation, kleene star, etc. For the alphabet $\Sigma = \{0, 1, *, (,), +\}$, the set of simplified state expressions is described by the predicate logic formula,

$$\phi = \neg\exists x \exists y (y = x + 1 \wedge Q(0, x) \wedge \neg Q(*, y)).$$

where we define the relation on the predicate $Q(\Delta, x)$ as: at position x, the expression contains the symbol Δ. Then, the predicate logic formula says, expressions do not contain 0 without $*$ next to it. For any given alphabet Σ where $|\Sigma| = 2$, a bijection $f : \Sigma \to \{0, 1\}$ for loading semantics to symbols, where 0 represents no state change, and 1 means state change, we say that there exists a pushdown automaton P with an alphabet $\Sigma' = \{0, 1, *, (,), +\}$ that decides whether the expression is in SCL.

7.2 Formulas, Models, and Theories

In this part, we give some important definitions related to formal systems and formal theories which will be used later on. Formalism is the idea of making a notion precise by using symbolic logic so that everyones agree on it. The logic itself is the system that allows us to formalize theories, deduce sentences from the axioms to obtain new theorems, etc. This is what we call *formal systems*. A formal system has the following components:

- an *alphabet* of symbols,
- a *grammar*,
- a set of sentences, the *axioms*,
- a finite set of *deduction rules*.

A *well-formed formula* is a syntatically correct expression in a formal language or in a formal language class. For example, if we consider the set of regular expressions as the set of all well-formed formulas over regular languages, then we can say that each

expression is a well-formed formula over the regular set. We will use this notion later. Regarding models in propositional logic, let ϕ be a propositional logic formula. Then, a *model* $\mathscr{M}$ is a true/false valuation on propositional atoms such that $\mathscr{M}$ satisfies ϕ which is denoted by $\mathscr{M} \models \phi$. In general, a model is an interpretation which makes the sentence true. A sentence is *satisfiable* if it has a model, and it is tautology, i.e. valid, if it is true in all models. Then, given a model $\mathscr{M}$, the set of sentences for which $\mathscr{M} \models \phi_1, \phi_2, \ldots, \phi_n$ is called the *theory of* $\mathscr{M}$, denoted as $Th(\mathscr{M})$. $\mathscr{M}$ is called the *axioms* of $Th(\mathscr{M})$. Given a theory T, if some sentence S is provable in T, we denote it by $T \vdash S$. A theory T is *consistent* if there exists no sentence $A \in T$ such that both $T \vdash A$ and $T \vdash \neg A$ holds. We say T is *inconsistent* otherwise. An example of an inconsistent set would be $\{S, \neg S\}$. A formal theory is said to be *complete* if every true sentence is provable. If G and T are two formulas and $G \models T$, then $G \cup \neg T \models \emptyset$, i.e. inconsistent (or contradiction). Let $G = \{\phi_1, \phi_2, \ldots, \phi_n\}$ be a set of well formed formulas. Finally, a theory T is *decidable* if for any given sentence A it is possible to determine whether or not $A \in T$. In propositional logic, given a formula ϕ we can decide whether or not ϕ is satisfiable by writing the truth table for the formula. For n atoms, we therefore have 2^n, hence finite, possible valuations. Before giving the adequacy theorem [15] for propositional logic which states that for any sentence S, if S is a theorem then S is a tautology, and if S is a tautology then S is a theorem, let us give two lemmas first.

Lemma 4. If S and $S \rightarrow T$ are both tautologies, then T is a tautology.

Proof. Assume that T is not a tautology. Then, given the fact that S and $S \rightarrow T$ are tautologies, there must be a model such that $v(S) = \top$, $v(S \rightarrow T) = \top$, and $v(T) = \bot$ [1]. If $v(S \rightarrow T) = \top$, then $v(S) = \bot$ or $v(T) = \top$ or both. If $v(S) = \bot$, this contradicts with $v(S) = \top$. If $v(T) = \top$, then it contradicts with $v(T) = \bot$. In either way we get a contradiction. Therefore, T must be a tautology. ■

Lemma 5. Let S be any sentence and let $S_1, \ldots, S_n$ be the propositional variables of S. Let v be any model. Then, if $v(S_i) = \top$ let S'_i be S_i while if $v(S_i) = \bot$ let S'_i be $\neg S_i$. Also let S' be S if $v(S) = \top$ and S' be $\neg S$ if $v(S) = \bot$. Then we have $\{S'_1, \ldots, S'_n\} \vdash S'$.

The adequacy theorem is often called as soundness and completeness. Soundness

[1] For some variable p, $v(p)$ means the logical value of p. $\top$ means true, and $\bot$ means false.

means every provable statement is true. Completeness means every true statement is provable. Now we give the proof of soundness, than later on completeness, thus we get the proof of the adequacy theorem.

Theorem 10. *(Soundness)* For any sentence S, if $\vdash S$ then $\models S$.

Proof. If S is a theorem, then there is a sequence $S_1, \ldots, S_n$ with $S_n = S$ such that the sequence is a proof of S. If S is an axiom, then there is nothing to prove since its truth value is known to be true and can be shown in a straightforward manner. If S is not an axiom, then we can have an induction on the length of the proof of S. If $n = 1$, then S is an axiom, so S is a tautology. If $n > 1$, then we assume that all theorems with proofs of length less than n are tautologies. So, either S is an axiom or the proof of S contains two sentences S_i and S_j, with $i, j < n$, such that S_j is, say, $(S_i \to S)$. S_i and S_j are tautologies by hypothesis, and by Lemma 4, S is a tautology too which completes the proof. ■

Theorem 11. *(Completeness)* For any sentence, if $\models S$ then $\vdash S$.

Proof. Assume that $\models S$. Let $S_1, \ldots, S_n$ be the propositional variables in S. By Lemma 5 we know that, for any model, $\{S'_1, \ldots, S'_n\} \vdash S$. Note that S' is S since $\models S$, so for any model S is true. Therefore, we have $\{S'_1, \ldots, S'_{n-1}, S_n\} \vdash S$ and $\{S'_1, \ldots, S'_{n-1}, \neg S_n\} \vdash S$. Using the deduction theorem we have $\{S'_1, \ldots, S'_{n-1}\} \vdash S_n \to S$ and $\{S'_1, \ldots, S'_{n-1}\} \vdash \neg S_n \to S$. Since we can show $\vdash (S_n \to S) \to ((\neg S_n \to S) \to S)$ we have, using Modus Ponens twice, $\{S'_1, \ldots, S'_{n-1}\} \vdash S$. By repeating this process $n - 1$ more times we have that $\vdash S$ as required. ■

Assigning true false valuations to variables is not the case in predicate logic. First let us define the elements of predicate logic. A language of predicate logic based on $\langle P, U, F\rangle$, denoted as $L(\langle P, U, F\rangle)$, is defined by the alphabet consisting of *connective symbols* $\{\neg, \to, \wedge, \vee, \leftrightarrow\}$, a set of *quantifier symbols* $\{\forall, \exists\}$ where $\forall$ is the universal quantifier and $\exists$ is the existential quantifier, a set of *punctuation symbols*, a set $P = \{P_1, P_2, \ldots, P_n\}$ of $n-$*ary predicates*, a *universe* (also called *names* set $U = \{a_1, a_2, \ldots\}$, and a set $F = \{f_1, f_2, \ldots, f_n\}$ of $n-$*ary function symbols*. We also have a set of *variables* used in the formulas. For instance, in $\phi = \forall x P(x)$, x is a variable bounded by the universal quantifier, and $P(x)$ is the scope of it.

Definition 15. A variable x in a formula ϕ is *free* if x is not bounded by any quantifier.

We said that true false valuations do not give us the truthness of the predicate logic sentence. Rather, we have interpretations. At this point we should consider the semantics of well formed formulas in predicate logic. In predicate logic an *interpretation* of a language $L = L(\langle P, U, F\rangle)$ over a universe U is a pair $I_U^L = \langle \phi_U, \psi_U \rangle^L$ where, $\psi_U : P_n \to \Re U^n$ maps, for each $P_n \in P$, the $n-ary$ predicate symbols to $n-ary$ relations over U.
$\phi_U : F_n \to (U^n \to U)$ maps, for each $F_n \in F$, the $n-ary$ functions symbols to $n-ary$ functions over the universe.
An *assignment* (also called model) is one instance of an interpretation, and we have the following properties:
Let S and G be any formulas. Then,

1. $\mathscr{M} \models \neg S$ iff $\mathscr{M} \not\models S$,
2. $\mathscr{M} \models (S \wedge G)$ iff $\mathscr{M} \models S$ and $\mathscr{M} \models G$,
3. $\mathscr{M} \models (S \vee G)$ iff $\mathscr{M} \models S$ or $\mathscr{M} \models G$, or both,
4. $\mathscr{M} \models (S \to G)$ iff $\mathscr{M} \models (\neg S \vee G)$,
5. $\mathscr{M} \models (S \leftrightarrow G)$ iff $\mathscr{M} \models ((S \wedge G) \vee (\neg S \wedge \neg G))$,
6. $\mathscr{M} \models \exists x S$ iff there is an element $a \in U$ such that there is some assignment that makes S true.
7. $\mathscr{M} \models \forall x S$ iff $\mathscr{M} \models \neg\exists x \neg S$. That is, there is no assignment that makes $\neg S$ true.

It is now worth giving the following definition.

Definition 16. A predicate logic formula ϕ is *satisfiable* is there is some model $\mathscr{M}$ such that $\mathscr{M} \models \phi$. A formula is *valid* if it is true for every model. A formula is *unsatisfiable* if there is no model that makes it true.

One important question about the predicate logic is regarding the validity of a given formula. That is, given a predicate logic sentence S, is S valid? It is crucial to note that this is an undecidable problem. This is one of the most fundamental results

in mathematical logic. In propositional logic, the answer to this question is easy. We browse all possible valuations and see whether one of them satisfies the given sentence. However, this is not the case in predicate logic since there are infinitely many models.

7.3 The Axioms

In this section, an axiomatic treatment to abstract power consumption over finite automata and SCL is given, based on state expressions. Formalization of these notions are required both for a better understanding, and for the next section related with the satisfiability. We have several elementary axioms. But first let us define the language of this theory.

Let Σ be the set of all state expressions in the correct form that are generated from $\{0, 1, +, \circ, *, (,)\}^*$. The symbol $+$ is an commutative and associative binary operator for union of two state expressions, $*$ is a unary operator for kleene star, and $\circ$ is an associative binary operator for the concatenation of two state expressions. For convenience, we do not write the concatenation operator explicitly. We use first-order logic, and we define $U = \Sigma \cup \mathbb{N} \cup \Gamma \cup \{\mathbf{0}, \mathbf{1}\}$ as the set of universe, where Γ is the set of all finite automata structures, and **0**,**1** are constant symbols intrepreted as state change and no state change, respectively. They should not be confused with the numerals in $\mathbb{N}$. We define $I(E, a)$, $Q(E, a, i)$, $>(A_1, A_2)$, $M(E_1, E_2)$, where $a \in \Sigma$, $E \in U$, and $i \in \mathbb{N}$ as relations, i.e. predicates. $I(E, a)$ is interpreted as; the expression E includes the symbol a. $Q(E, a, i)$ is interpreted as; the *ith* position of the expression E is the symbol a. We interpret $>(A_1, A_2)$, shortly $>$ as; the minimum abstract power consumption of the automaton A_1 is more than that of the automaton A_2. $M(E_1, E_2)$ means, E_1 contains more 1's than that E_2, where E_1 and E_2 are state expressions. We define $A(E)$ and $S(E)$ as functions. $A(E)$ gives the automaton structure whose state expression is E. Finally, we interpret $S(E)$ as the number of states of the automaton whose state expression is E.

1. Axiom of distinction: $0 \neq 1$. 0 and 1 symbols are not equal in either syntactic or semantic (i.e. constraint of at least one of the symbols should stand for state

change, and the other one opposite) meanings.

2. Axiom of equality: $\forall x \forall y (M(x,y) \iff A(x) > A(y))$. This axiom states that the abstract power consumption of $A(x)$ is greater than that $A(y)$ if and only if x has more 1's than y has.

3. Axiom of non-zero: $\neg \exists x (j = i+1 \wedge Q(x,0,i) \wedge \neg Q(x,*,j))$. This statement says the similar thing that we have said about the restrictions of state expressions. There is no such state expression that contains a star-free 0 symbol. This axiom is not known to be true for standard SCL description, but true for ϵ-SCL.

4. Axiom of transitivity: $\forall x \forall y \forall z ((M(x,y) \wedge M(y,z)) \rightarrow M(x,z))$. That is, if x has more 1's than y, and y has more 1's than z, then x has x has more 1's than z which means, by axiom of equality, the abstract power consumption of the the automaton whose state expression is x is greater than the abstract power consumption of the automaton whose state expression is z.

5. Axiom of succession: $\forall x \exists y (A(y) > A(x))$. This axiom states that there is always some finite automaton which consumes more abstract power than any other finite automaton.

6. Axiom of singularity: $\forall x (I(x,1) \rightarrow S(x) > 1)$, where $>$ is the usual *greater than* operator defined over numbers. We say that for any state expression x, if x contains the symbol 1, then the automaton whose state expression is x has at least two states.

7. Axiom of uniqueness: $\exists! x \neg I(x,1)$. That is, there exists a unique state expression such that it does not contain any 1. Consequently, this expression is 0^*.

We now give two definitions in order to say something about the decidability of the relations used in the axioms, given above.

Definition 17. A *total function* is a function $f : \Delta \rightarrow \Gamma$ which maps every element of $x \in \Delta$ to exactly one corresponding value $f(x) \in \Gamma$. We say that the *range* of the function $f : \Delta \rightarrow \Gamma$ is $\{f(x) \mid x \in \Delta\}$, i.e. the set of elements in Γ that are the values of f for arguments in Δ.

Definition 18. An n-ary relation $R(a_1, a_2, \ldots, a_n)$ is *effectively decidable* iff there is an algorithmic procedure that could decide, in a finite number of steps, whether the relation applies for any given case.

Definition 19. A total function $f : \Delta \rightarrow \Gamma$ is *effectively computable* iff there is an algorithmic procedure that could calculate, in a finite number of steps, the value of the function, $f(x)$, for any given $x \in \Delta$.

The term *effective* is important here. An effective procedure [16] involves executing an algorithm that successfully terminates for all instances. Since we can think of algorithms as Turing computable functions, by Church-Turing Thesis, an effective procedure is one that a Turing machine computes. So, effective decidability or computability does not depend on the architecture. That is, algorithmic decidability and computability is architecture independent. Before we discuss the satisfiability of state change expressions on abstract power consumption, let us first give the following definitions on enumerability. We say that a set is enumerable if it is possible, in principle, to enumerate its elements. That is, there must be some function that counts its elements in some order. For example the set $\{0, 1, 2, \ldots, 9\}$ is clearly enumerable because it is finite. So, every finite set is enumerable. If a set S is infinite, such as the set of all integers, then we must find a function which maps every element of $\mathbb{N}$ to some element in S. In this case, S is called *denumerable*. If no such mapping exists, then the set is called *nonenumerable* or simply *uncountable*. We have already given an uncountable set, that is the set of all binary sequences, in Section 5. Note that the set of real numbers is also uncountable. Hence, we give the followings.

Definition 20. A set Σ is *enumerable* if and only if either Σ is empty, or there is some surjective function $f : \mathbb{N} \rightarrow \Sigma$.

Definition 21. A set Σ is *effectively enumerable* if and only if either Σ is empty, or there is some effectively computable function that enumerates it.

Note that enumerability and effective enumerability are different. Every effective enumerable set is clearly enumerable. But not every enumerable set is effective enumerable. There are infinite sets that are enumerable, but cannot be effectively enumerated such as the set of arithmetical truths. In fact, this is Gödel's First Incompleteness Theorem [16].

7.4 Axiomatized Formal Theories

Mathematical logic is concerned with some philosophical questions. What is logic? Why do we need formal axiomatic systems? Why do we need formalism at all? In logic classes, we are asked to translate arguments into an appropriate formal language, and then find formal proofs from given premises. The purpose of using formal languages is because natural languages contain redundancies and ambiguities. Using an artificial language can eliminate these problems and as well as the contradictions in mathematics. This was what David Hilbert proposed as a goal at the beginning of the twenieth century in a lecture given at the International Congress of Mathematicians. Eliminate all the problems and contradictions in mathematics by formalizing all of mathematics using a formal axiomatic system with an artificial grammar, a set of axioms and etc. Gödel and Turing, however, showed that it could not be done. That is, there is no complete and consistent formal axiomatic theory which can capture all truths in mathematics. The failure of Hilbert's programme surprisingly led to the creation of theoretical computer science. In a sense, computers are the instances of formal systems. This isomorphism can be explained as follows [14].

Formal System	**Digital Computer**
Alphabet of the language	bits of 1's and 0's
Well-formed formulas	Bytes (8-bit strings)
Grammar of the language	Instruction format
Axioms	Input
Deduction rules	Program
Theorems	Output
Proof	Computation

Notice that proofs really correspond to computations. Therefore, limits of provability implies limits of computation. However, we can say that provability refers to a larger class. A universal Turing machine is one that is capable of simulating any other Turing machine. Because of the universality, computability is a complete theory. But this is not the case in mathematics. There is no complete universal formal system for mathematics. Even the most powerful theory of mathematics which is considered to

be the Zermelo-Frankael Set Theory, cannot prove the set of all arithmetical truths.

We said that formal systems consist of a formal language, a set of axioms, and deduction rules. We will focus on the decidability issues. First of all, for any formal axiomatized theory, we must be able to effectively decide whether or not a given expression is a well-formed formula. We are using first-order logic as the language of the theory. The given expressions is finite. Therefore, one can check if it is in the correct form or not. Secondly, we must be able to effectively decide whether or not a given well-formed formula is an axiom. In our case, the set of axioms is finite which makes this property decidable. This property does not rule out theories with infinitely many axioms, because one may have a set of axioms in a certain form. That is, if one asks that given a well-formed formula ϕ, whether or not ϕ is an axiom, it is decidable if the axioms are in a certain form. So, one can just check its syntactic form, and decide the problem. An axiomatized formal theory also needs some deduction rules. As it can be seen, we use state change expressions combined with the first-order logic. Finally, in formal axiomatized theories when we are given a true sentence, say ϕ, which is also called a theorem, it can be traced back to the set of axioms. That is, if we have a formal axiomatized theory T, and if we deduce a sentence ϕ, we say that ϕ is a theorem of the theory, and it is provable. We indicate this as $T \vdash \phi$. We are now ready to give a simple example to illustrate some of the definitions that we have given related to formal systems and formal axiomatized theories.

Example 2. Consider two formal theories T_1 and T_2, whose language consists of the set of propositional atoms $\{p, q, r\}$ together with all the well-formed formulas that can constructed from them using the propositional connectives and operators, whose logic is a standard natural deduction system for propositional logic, and whose sets of axioms are $\{\neg p\}$ and $\{\neg p, q, \neg r\}$, respectively. Therefore, T_1 and T_2 are both formal axiomatized theories. Both theories are consistent. That is, it is not the case that we prove S and $\neg S$ from the theory, where S is an arbitrary sentence in the theory. Moreover, both theories are decidable. That is, given a sentence which belongs to the language of that theory, we can determine whether or not it follows from the axioms. One important thing is that although T_1 is a decidable theory, this does not mean that it decides every well-formed formula. For instance, T_1 does not decide the well-formed formula $(q \wedge r)$. Because the axioms of T_1 do not entail either $(q \wedge r)$ or

$\neg(q \wedge r)$. So, completeness of a theory, and the capability of checking mechanically what a theorem is, are different from each other. On the other hand, T_2 is a complete theory, i.e. we can prove, for any given sentence ϕ in the language of T_1, either $T \vdash \phi$ or $T \vdash \neg\phi$.

Now there is one last thing we should discuss about the enumerability of formal theories. We next show that every axiomatized formal theory has some properties, and we state this fact with the following theorem.

Theorem 12. If T is an axiomatized formal theory, then (i) the set of well-formed formulas, (ii) the set of proofs constructible in T, (iii) the set of theorems are effectively enumerable.

Proof. (i) If T is a formal theory, then it has a finite set of alphabets, say Σ. We can have an algorithm that generates all possible strings, i.e. Σ^* in a lexicographic order. That is, first strings of size 1, then 2, and so on are generated. It is clearly countable infinite. Let us call the set of all well-formed formulas as W. Obviously, $W \subseteq \Sigma^*$. By definition of formalized languages, there is a mechanical procedure to check which of the strings count as well-formed formulas. So, for every $w \in \Sigma^*$ we check whether w is a well-formed formula or not. By this way we can effectively enumerate the set of all possible well-formed formulas, i.e. the set W.

(ii) Similar to enumerating the set of possible well-formed formulas, we can enumerate all proofs in the language of T in size order. Since proof is a finite sequence of well-formed formulas, the set of all proofs in T is countable infinite. In a brute-force fashion, we enumerate all possible finite sequences of well-formed formulas. By definition of axiomatized theory, there is an algorithm to check whether or not a sequence of well-formed formulas is a valid derivation (proof) from the axioms of T. So, we can select those are valid derivation which eventually gives us the effective enumeration of the set of all possible proofs in the language of T.

(iii) In this part, we start by enumerating the set of all possible well constructed proofs, then check if the conclusion is satisfiable by the axioms. ■

It is important to note that effective enumerability of the theorems in an axiomatized formal theory does not imply the theory is decidable. Given an arbitrary formula ϕ,

if ϕ is really a theorem in a formal axiomatic theory T, then it will eventually appear on the enumeration list. However, we may not able to decide the other way.

The relations and functions, used as non-logical symbols in the axiomatization of abstract power consumption, are decidable. There is an effective procedure that always halts. Moreover, the theory of abstract power consumption is complete. It is complete due to the fact that, in first-order logic, the theory of any structure is a complete and consistent set of sentences [30].

Theorem 13. For any structure M in the language of the theory, $Th(M)$ is a complete and consistent theory.

Proof. We have to show that for any structure M, and for any sentence ϕ; either $\phi \in Th(M)$ or $\neg\phi \in Th(M)$ but not both. It would be inconsistent otherwise. Since $\neg$, $\wedge$, and $\exists$ are universal symbols for first-order logic and are capable to define other logical symbols, we assume that ϕ does not contain any symbols other than $\neg$, $\wedge$, and $\exists$. The proof is by induction on the total number of occurences of $\neg$, $\wedge$, and $\exists$ in ϕ.
Basis: If ϕ contains none of the universal symbols, then this implies that ϕ has the form $R(t_1, \ldots t_n)$ or $t_1 = t_2$ where $t_1, \ldots, t_n$ are terms defined in the language of the theory. Simply, ϕ is atomic. Since ϕ is a sentence, it is variable-free. That is, ϕ is not bounded by any quantifier. Since, M is a structure defined by the language of the theory and each t_i is a variable-free term, M interprets each t_i as an element a_i of the universe U of M. By the definition of $\models$, $M \models t_1 = t_2$ if and only if a_1 and a_2 are the same element of U. Similarly, $M \models R(t_1, \ldots, t_n)$ if and only if $(a_1, \ldots, a_n)$ is in the a subset of U^n that the interpretation of M assigns to R. In each cases, $M \models \phi$ or $M \models \neg\phi$ but not both. This completes the basis case.
Induction: Our inductive hypothesis is that we assume for any sentence containing at most m occurences of $\neg$, $\wedge$, and $\exists$, this property holds. Suppose that ϕ has the form $\neg\psi$ or $\psi \wedge \theta$. Then, by inductive hypothesis, the property holds for both ψ and θ. By the semantics of $\neg$ and $\wedge$, the property also holds for ϕ. Finally, suppose that ϕ has the form $\exists\psi(x)$. Similarly, by the semantics of $\exists$, $M \models \phi$ if and only if $M_C \models \psi(c)$ for some constant c in the language of M_C. By inductive hypothesis the property holds for $\psi(c)$. Therefore, it also holds for ϕ which completes the proof. ■

Here, our structure is a two tuple $A = \langle U, \Psi \rangle$ where Ψ is the interpretation that was given. Due to the above theorem, we say that $Th(A)$ is complete.

7.5 Satisfiability of Finite Automata Properties

In this part, we show that the formalism that we have constructed can be used in the satisfiability of state expressions, when given a formal specification related to abstract power consumption along with a finite automaton or regular expression. The given formal specification, in first-order logic, and regular expression which is converted to its state expression is tested to satisfy another expression under that specification. So, we show how to satisfy the properties of state expressions under given formal specification. For example, if one has a simple computational model that exhibits a state change characteristic, and want to find similar models, we define a formal specification, which has to be the property of that computational model, then test whether another given candidate model satisfies the property. By this way, we conclude the equivalence of two abstract models with respect to the given formal specification. This means that, both models have the same behavior according to our specification. It is worth to note that the domain set that we are working on is a countable set.

Let us define state expressions as the set of all well-formed formulas in SCL. Call this set Ψ. Then,

$$\Psi = \bigcup_{i=1}^{\infty} L_i = L_1 \cup L_2 \cup \cdots$$

Notice that Ψ is countably infinite, hence it is denumerable. That is, we can assign a unique natural number to each state automaton or its equivalent state expression. The enumerability of well-formed formulas is usually called Gödel numbering which is originally used in the proof of Gödel's incompleteness theorem. Suppose we are given a regular language R (that we want to find a model for), a state change language $\mathscr{M}$, and a formal specification Γ written in first-order logic. Then, we can decide whether or not $\mathscr{M}$ satisfies R under the formal specification Γ, we write as $\mathscr{M} \models_\Gamma R$. The question is "does the given candidate model $\mathscr{M}$ satisfy our state automaton $L_i = C(R)$, hence R, with respect to the specification Γ"? We say that $\mathscr{M} \models_\Gamma L_i$ iff $\mathscr{M} \models \Gamma$ and

$L_i \models \Gamma$. That is, if both the state expression of a given finite automaton and the given candidate model $\mathscr{M}$ satisfy the specification Γ, then we say that $\mathscr{M}$ satisfies the state expression of the given finite automaton under Γ. Now, we give the following definition.

Definition 22. A language M is *equivalent* to $L_i \in \Psi$, denoted as $M \cong_\Gamma L_i$ with respect to a specification Γ, whenever $M \models_\Gamma L_i$.

Consider the following example.

Example 3. For convenience let us give a simple one. Suppose that we are given a finite automaton $A = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1)$, where $\delta(q_0, a) = \{q_0\}$, $\delta(q_0, b) = \{q_1\}$, $\delta(q_1, b) = \{q_1\}$. Then, $F = C(A) = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, q_1)$ such that $\delta(q_0, 0) = \{q_0\}$, $\delta(q_0, 1) = \{q_1\}$, $\delta(q_1, 0) = \{q_1\}$. Clearly, $L(F) = 0^*10^*$. Let us define our specification as follows:

$$\Gamma = \neg\exists x \exists y (y = x + 1 \wedge Q(1, x) \wedge Q(1, y)).$$

It consists of the set of all finite automata that do not change their state consecutively on any input. Then, one may give a candidate model $M = 10^*10^*$. We can conclude that $M \models_\Gamma F$, because M does not change its state consecutively. Hence, we say that $M \cong_\Gamma F$. However, if $M = 110^*1$, then $M \not\models_\Gamma F$.

To summarize; the aim is to check the abstract power consumption relation between two given finite automata. We must define a formal specification Γ expressed in first-order logic, along with a finite automaton A such that the property about abstract power consumption for A is satisfied by Γ. We are given another finite automaton B to be checked whether the property of B is also satisfied by Γ. Then, we conclude that the properties of A and B are both satisfied by Γ. Hence, we say that A and B are equivalent with respect to Γ.

This method is one formal approach to satisfy the property in terms of abstract power consumption of some finite automaton, when given a formal specification written in first-order logic. This simple formal method can be used, as explained before, to infer about the abstract power consumption relationship between two computational models, particularly to find equivalent models under the given specification.

Note that this method should not be confused with model checking. In model checking, we have a sentence and we look for some valuation or interpretation, depending on the logic, which makes that sentence true. Here, although the specification is written in the form of first-order logic, the state expression is not an interpretation. Instead, it is a mathematical object that has the property of what formal specification expresses.

GCRIS

CHAPTER 8

CONCLUSIONS

A new proper subset of regular languages, called state change languages, is defined and related descriptions: state expressions and automata are given in this thesis. The language generated by the output automaton can be considered as an *"internal"* language defined by state transitions of input automaton during its computation. The obtained class is shown to be closed under union and intersection, but not under complementation. From our language morphism result, one can talk about the existence of a language morphism between RL and SCL. Then, the decidability of SCL is discussed. We also introduced the notion of abstract power consumption which we think is an important concept for defining lower bounds for the abstract energy needed for the computation of some model, particularly finite automata, for all inputs. Then, we related Chaitin's work on irreducible structures to our problem of constructing a particular uncomputable number. We then gave an axiomatization of the notion of abstract power consumption over finite automata with an application to satisfiability.

This thesis has two results. The work in this thesis is the first step towards a universal analysis on algorithmic energy complexity of models of computation. Second result is the existence of a relationship between the formal language set of the model of computation to which the operator is applied, and the language of the output description. One open problem is the algebraic characterization of state change languages in which we expect a similar result as that of the case in W-languages. More importantly, in the future, one can define such an operator, exhibiting the abstract power consumption characteristic of the abstract model, for other computational models, especially those which are not well defined physically, in order to have an idea about their lower bounds on power consumption characterized by their mechanics. Such

an example is the lower bound for abstract power consumption of quantum Turing machine. It would be invaluable if one can specify a lower bound for some core computations used in algorithms such as quantum fourier transform, etc. Although some algorithms require a polynomial time, they may require enormous amount of energy. To define lower bounds for algorithmic energy complexity, one must define a universally accepted operator for quantum Turing machine to encode its abstract power consumption characteristic in order to deduce a formal result.

GCRIS

REFERENCES

[1] A.Çevik, H.Kılıç: On an Infinite Proper Subset of Regular Languages, Technical Report, TR-ATU-2009/07-001 (2009)

[2] H.Kılıç: On the Power Consumption of Digital Computing Systems. IEEE 13. Sinyal İşleme ve İletişim Uygulamaları Kurultayı (SIU 2005) Bildiriler Kitabı (2005)

[3] A.C.Gómez, J.-E.Pin: A Robust Class of Regular languages. MFCS 2008, LNCS 5162, pp. 36-51 (2008)

[4] P.A.Abdulla, B.Jonsson, M.Nilsson, and M.Saksena: A Survey of Regular Model Checking. International Conference on Concurrency Theory No.15, London, England, vol.3170, pp. 35-48 (2004)

[5] E.M.Gold: Language Identification in the Limit, Inform. Contr., 10,447-474 (1967)

[6] D.Angluin: Inference of Reversible Languages. J. ACM 29, 741-765 (1982)

[7] N.Tanida, and T.Yokomori: Polynomial-Time Identification of Strictly Regular Languages in the Limit. IEICE Trans. Inf & Syst. E75-D, 125-132 (1992)

[8] J.D.Emerald, K.G.Subramanian, and D.G.Thomas: Learning Code Regular and Code Linear Languages. In: Proceedings of ICGI-96, Lecture Notes in Artificial Intelligence 1147, 211-221 (1996)

[9] E.Mäkinen: Inferring Uniquely Terminating Regular Languages From Positive Data. Information Processing Letters, vol. 62, Issue 2, pp. 57-60 (1997)

[10] M.Sipser: Introduction to the Theory of Computation, Thomson Course Technology, 2nd Ed. (2005)

[11] J.E.Hopcroft, R.Motwani, J.D.Ullman: Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 2nd Ed. (2001)

[12] K.Gödel: On Formally Undecidable Propositions in Principia Mathematica and Related Systems I (1931)

[13] G.J.Chaitin: The Limits of Mathematics, A Course on Information Theory and the Limits of Formal Reasoning, Springer-Verlag, London (2003)

[14] P.Suber: Formal Systems and Machines, http://www.earlham.edu/ peters/courses/logsys/machines.htm

[15] S.Reeves, M.Clarke: Logic for Computer Science (International Computer Science Series), Addison Wesley (1990)

[16] P.Smith: An Introduction to Gödel's Theorems, Cambridge University Press (2007)

[17] M.Jehle, J.Johannsen, M.Lange, N.Rachinsky: Bounded Model Checking for All Regular Properties, Proceedings of the Third International Workshop on Bounded Model Checking, ENTCS 2005 (2005)

[18] S.Yovine: Model Checking Timed Automata, European Educational Forum: School on Embedded Systems, p.114-152 (1998)

[19] George H.Mealy: A Method for Synthesizing Sequential Circuits, Bell Systems Technical Journal 34, p.1045-1079 (1955)

[20] Edward F.Moore: Gedanken-experiments on Sequential Machines, Automata Studies, Annals of Mathematical Studies, Princeton, N.J: Princeton University Press, 34, p.129-153 (1956)

[21] A.M.Turing: On Computable Numbers with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Ser.2, Vol.42 (1937)

[22] H.G.Rice: Classes of Recursively Enumerable Sets and Their Decision Problems, Trans. Amer. Math. Soc. 74, 358-366 (1953)

[23] R.Jain, D.Molnar, Z.Ramzan: Towards a model of energy complexity for algorithms, Wireless Communications and Networking Conference (WCNC 2005), IEEE, p.1884-1890, Vol.3 (2005)

[24] K.Zotos, A.Litke, A.Chatzigeorgiou, S.Nikolaidis, G.Stephanides: Energy Complexity of Software in Embedded Systems, Nonlinear Sciences - Adaptation and Self-Organizing Systems (2005)

[25] G.J.Chaitin: Algorithmic Information Theory, Cambridge University Press (2004).

[26] G.J.Chaitin: On the Length of Programs for Computing Finite Binary Sequences, Journal of the ACM (JACM), Vol.13, Issue 4, p.547-569 (1966)

[27] C.S.Calude: Simplicity via Provability for Universal Prefix-free Turing Machines, CDMTCS Research Reports CDMTCS-339 (2008)

[28] H.Straubing: On Logical Descriptions of Regular Languages. In: S.Rasjbaum (ed.) LATIN 2002. LNCS, vol. 2286, pp. 528-538. Springer, Heidelberg (2002)

[29] B.Khoussainov: On Algebraic and Logical Specifications of Classes of Regular Languages. Theoretical Computer Science, vol. 298, Issue 2, pp. 325-346 (2003)

[30] S.Hedman: A First Course in Logic - An introduction to model theory, proof theory, computability, and complexity, Oxford University Press (2006)

[31] P.Weil: Algebraic Recognizability of Languages. MFCS 2004, vol. 3153 of LNCS, pp. 149-175, Springer-Verlag (2004)

[32] P.Suppes: Axiomatic Set Theory, Dover Publications, Inc., New York (1972)

[33] N.Immerman: Descriptive Complexity (Graduate Texts in Computer Science), Springer (1999)

GCRIS