IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 28, Issue 3, Ser. II (May. – June. 2026), PP 29-34
www.iosrjournals.org

From Theory of Computing to Automata

Hieu D. Vu
Fort Hays State University
600 Park Street
Hays, KS. 67601

Abstract
The theory of computing explores the fundamental computing capabilities of computers also their limitations. They focus on what types of problems can be solved and the effectiveness they can be addressed.
The subject, concept of Automata and Computing Theories was introduced in 1979, and in those early years, Automata, Computing Theory, and Language Theory were still listed as an area of active research, based heavily in Mathematics and largely a required course for senior or graduate students.
Keywords: *Automata, Computing theories, Formal language, Science, Technology, Computing machines, Turing machines, Mathematics, Algorithm.*

Date of Submission: 28-05-2026 Date of Acceptance: 08-06-2026

I. INTRODUCTION

In theoretical computer science and mathematics, the Theory of Computing is a branch that studies what type of problems can be solved on a model of computation using algorithms, and how efficiently they can be solved and to what degree (approximation to the precisely solution). Generally, the Theory of Computing is a branch of theoretical computer science that concentrates on some important areas following:

- Automata Theory: This branch studies the abstract machines and what they can do in solving problems. Automata included finite automata, context free grammars, and the imaginary Turing machines to demonstrate how an ideal machine performs computation.
- Computability Theory: This branch of computing deals with what types of problems that can be or cannot be solved and the decidability, limits of algorithmic solutions.
- Computational Complexity Theory: This area classifies problems based on their inherent of difficulty and the resources required to solve them (time, space…). It examines problems of classes such as P and NP. [1]

II. COMPUTING THEORY

In three principle areas of theory of computing: Automata, Computability, and Complexity. They are linked by the central question: What are the fundamental capabilities and limitations of computing machines (computers)? This question went back to the decade of 1930s when mathematicians started to find the meaning of computation. Since that time, technology advanced rapidly, increased the ability of our computing and moved this question from the theory into the real world of practical concern.

AUTOMATA THEORY: concentrates on the definitions and properties of mathematical models of computation. These models applied in some areas of computer science. One of these model called “Finite Automaton” is used in text processing, compilers and hardware designs. Another model called “Context-Free Grammar” is used in Programming Languages and Artificial Intelligence (AI).
Automata theory is good as the starting point for the study of theory of computation. The theories of computability and complexity need a well-defined of a computer for processing data. Automata allows us to practice with formal definitions of computation.

COMPUTABILITY THEORY: In the early years, mathematicians Kurt Godel, Alan Turing, and Alonso Church found that certain basic problems cannot be solved by computing machines (computers). One example of this problem is to determine if a mathematic logical statement is true or not (false). Logic is very important in mathematics, but at that time, no computer algorithm can perform this task.
Result of this finding led to the development of ideas of theoretical models for computing machines that would help in the making of actual computers.

The theories of computability and complexity are related. In the theory of complexity, objective is to classify the problems, which ones are solvable, others are not. Computability theory presents several concepts used in complexity theory.

COMPLEXITY THEORY: There are many different varieties problems for computer to solve, some are simple, easy, but others might be hard to solve. The different varieties among problems called complexity. For example, sorting thousands of numbers into ascending order is a simple problem that can be solved by a small, lab-top computer, but the scheduling problem such as class scheduling or even a lot more complicate like airline-reservation problem is much harder to solve even for a large, main frame computer, or super computer. [2]

II. 1. COMPUTABILITY

Can a computing machine solve any problems? Before answering this question, we need a programmer to write a program for the computer to do the computing job, and what if the programmer or our-self (human) cannot solve that particular problem because it is too hard, too difficult. Computing theory attempts to classify classes of problems to be solved by machines. What type problems can be solved, and what are not?

In the 1930s, mathematicians Godel, Turing, and Church found that some mathematical problems cannot be solved by a computing machine (computer). So the principle question for computability is: classifying problems which are solvable or unsolvable. And to answer the question, we need to know:

- The machine that perform the computing (computer, how good, reliable …)
- Algorithms (step by step to solve the problems), and
- Computation (the actual works on problems)

II. 2. COMPLEXITY THEORY

The main question for complexity is “What makes some problems are more difficult to solve then the others?” Typically, a problem is easy or simple is solvable efficiently, the other difficult or harder problems cannot solve efficiently, or we cannot determine they are solvable or un-solvable.

Examples: To solve an easy problem (solvable), we can program a computer to perform direct calculations or to follow a step by step algorithm that leads to the solution. The following algorithm used to sort a list of integers into ascending order.

```
SORTING ALGORITHM (numbers into ascending order)
for index i = 1 to n-1 do                    (outer loop)
    index contain the smallest number        (leading index contain the smallest)
    for index j = i+1 to n do                (inner loop to compare)
        if number[j] < number[i]             (number[j] < number[i]
            index = j                        (the new smallest indexed j)
    end for                                  (end inner loop, index j)
end for                                      (end outer loop, index i)
```

When the outer loop terminated, we have an ascending ordered a list of numbers.

Examples: of difficult or hard problems are classified as un-solvable or we don’t know for sure they are solvable or not or take a lot of efforts and resources (not efficiently). Some examples of this type of problems such as: courses scheduling at a university, computing for the layout Integrated Circuits chips Very Large Scale Integration (VLSI), or Airline Reservation System. [3]

III. AUTOMATA THEORY

Automata theory is a branch of mathematics and computer science that focuses on the study of abstract machines (computing devices) and the computational problems that can be solved by those theoretical abstract machines. Before computer was built in the 1930s. Computer pioneer Alan Turing studied an abstract machine that has all capacities of today computers. He tried to describe the boundary between what problems a computing machine could solve and what it could not. His conclusions apply not only to his abstract Turing machines but also today computers.

In the next two decades, 1940s, 1950s, a number of researchers studied simpler types of abstract computing machines called “Finite Automata”. Originally, these automata proposed to model brain function that proved very useful for other purposes. In the late 1950s, another computer pioneer, linguist N. Chomsky began to study formal “Grammars”. The grammars studied are not for only machines, they are closed to human languages and automata and played important roles today in building software components and compilers designs. [4]

III. 1. FINITE AUTOMATA

Finite automata are abstract machines is used to recognize patterns from input sequence or string of characters. It is the basis for understanding regular languages, and having the following properties:

- Consist of states, transitions, input symbols, and process each input symbol one at a time.
- After processing the input, if it ends in an acceptance state, then the input (string of symbols) is accepted, otherwise rejected.
- Finite automata come in two forms: deterministic (DFA) or non-deterministic (NFA) automata. Both can recognize the same set of regular languages.
- Finite automata are used wide in text processing, compilers designs, and network communication protocols.

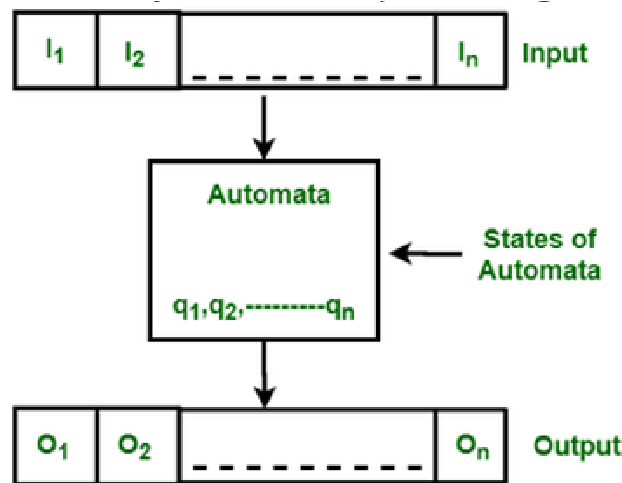

Figure: Features of Finite Automata

III. 1. 1. FORMAL DEFINITION OF FINITE AUTOMATA

A finite automaton (singular of automata) can be defined as a tuple:

$\{Q, \sum, q, F, \partial\}$

Where:

- Q: Finite set of states (S_1, S_2, S_3, …, S_n)
- $\sum$: Set of input symbols (1122… aaabb…)
- Q: Initial state
- F:Set of final states
- ∂: Transition function

III. 1. 2. FEATURES OF FINITE AUTOMATA

- Input: Set of symbols or characters input into the machine.
- Output: Accept or reject the input symbols base on the patterns of the input.
- States of Automata: Conditions or configurations of the machine.
- State Relation: The transitions between states according to the input symbols.
- Output Relation: Based on the final state (accepted or rejected).

III. 1. 3. TYPES OF FINITE AUTOMATA

There are two types of finite automata:

- Deterministic Finite Automata (DFA)

A deterministic finite automata (DFA) is a simpler one, defines as a tuple $\{Q, \sum, q, F, \partial\}$. In this type, for each input symbol, the machine transitions can only “one to one” state. DFA does not allow any null transitions, all states must have a transition defined for every input symbol.

DFA is defined by five tuples $\{Q, \sum, q, F, \partial\}$, where:

Q: set of all states

$\sum$: Set of input symbols (character string)

q: Initial state (starting state of the machine, usually set as q_0)

F: Set of final state

∂: transition function, defined as (∂: Q x $\sum$ ---> Q)

Example: Construct a deterministic finite automata (DFA) to accept all strings ending with character 'a' (symbol a).
Where: $\sum = \{a, b\}$, $Q = \{q_0, q_1\}$, $F = \{q_1\}$

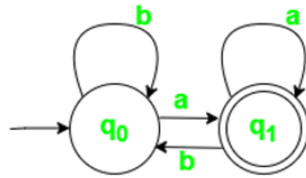

Fig 1. State Transition Diagram for DFA with Σ = {a, b}

Transition Table for the above Automaton

State\Symbol	a	b
q0	q1	q0
q1	q1	q0

If the string ends with character 'a', the machine reach state q_1 that is an accepting state (denoted as a double circle).

- Non-Deterministic Finite Automata (NFA)

NFA is similar to DFA but includes additional features:

a. The transitions can go to multiple states for one input symbol.
b. Null (€) transition is also accepted, where the machine can change states without an input symbol.

Example: Construct a non-deterministic finite automata (NFA) that accepts an input string ending with character 'a'
Where: $\sum = \{a, b\}$, $Q = \{q0, q1\}$, $F = \{q1\}$

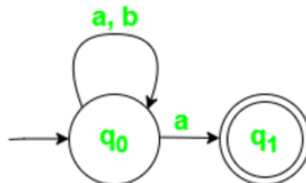

Fig 2. State Transition Diagram for NFA with Σ = {a, b}

State Transition Table for the above Automaton

State\Symbol	a	b
q0	{q0,q1}	q0
q1	φ	φ

In a non-deterministic automata (NFA), if any transition that leads to an accepting state (q_n), the input string is accepted. [5]

III. 2. APPLICATIONS OF FINITE AUTOMATA

1. Languages Acceptor

This is a simple computing device (machine), used to verify a string of characters (symbols) that belongs to the language or not. A string of characters is the input into the machine for processing one character at a time, and at

the end, the machine outputs “Yes” or “No”. “Yes” means it is accepted the string input or “The string is belong to the specified language”, “No” is otherwise.

Example 1: Let L_1 be the language defined by:

$L_1 = \{x \in \{a, b\}^* \mid x \text{ ends with aa}\}$

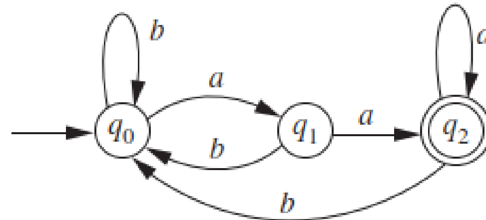

Figure 2.2 |
An FA accepting the strings ending with *aa*.

From the transition diagram above, q_0 is the initial state, input character ‘b’ doesn’t go anywhere (nothing progress), so the automaton stays in q_0. Input character ‘a’ let the automation go to state q_1. In q_1, input ‘a’ allows a transition to q_2, and the language L_1 which ending with “aa” is accepted (Q2 has a double circle, indicates accepting state), while ‘b’ will move it back to q_0 (L_1 is rejected).

Example 2: Let language L_2 be defined:

$L_2 = \{x \in \{a, b\}^* \mid x \text{ ends with b and does not include substring aa}\}$

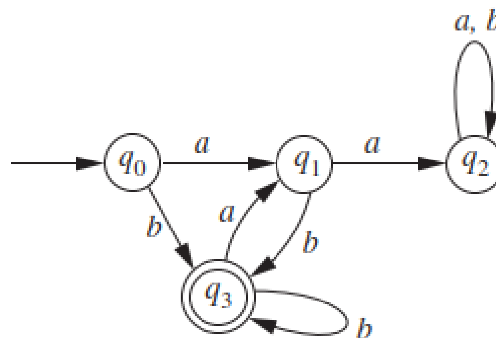

Figure 2.4 |
An FA accepting the strings ending with *b* and not containing *aa*.

In previous example, language L_1 is accepted whenever the last two inputs are both ‘a’, but in Example 2, if the next two inputs are both ‘a’ then the language L_2 is rejected (not accepting a substring “aa”). From this point, we want to make sure the automaton cannot reach the accepting state.

So we modify the diagram of L_1 in figure 22 above, the two consecutive inputs of character ‘a’ will stay in q_2 (not accepting state), and adding state q_3 which is the accepting state (double circle). The final, accepting state q_3 only receive character ‘b’ from q_0 or q_1. From initial state q_0, only character ‘b’ will move the automation to final accepting state q_3 (no character ‘a’), and from q_1 (already contain one ‘a’ then next character ‘b’ also move the transition to accepting state q_3 (no substring “a a”). [6]

2. Lexical Analysis

Compiler design for programming languages is one of important applications of automata. Among the tasks of a compiler is lexical analysis, which is the first phase that breaks down sequence of characters (source code, input) into meaningful tokens for further processing.

Lexical analysis is also called scanning, it is the initial step in the compilation process. In this phase, the lexical analyzer reads the input source code one character at a time (of cause, the speed of the computer is very fast), and groups these characters into smallest identities called tokens understand by the computers such as: keywords, identifiers, constants, operators, etc…
One of the tasks performed by the lexical analyzer is removing unnecessary elements for processing such as blank spaces and comments. This process also detects syntax errors in the source code, and diagnosis possible recommendations for the programmers to fix the errors. When all the errors are removed, the compilation process continues and executes the program that is already converted into machine readable forms and provide the outputs (results from the execution of the programs). [7]

IV. CONCLUSION

With the current technology, we are living in a rapid changing world. No question, the computer have changed the ways we do things, the way we learn, the way we do business, and almost everything, even entertaining. Theory of computing and automata contributed greatly in the development of our today computers. How a theoretical computing devices in the 1930s era performed computing, or processing input data, and from that day, mathematicians, computer pioneers kept improving. Today, with technology and computer industry advanced rapidly to build powerful computing, data processing machines or computers for us to use.
From that old theory of computing and automata, we can build applications such as:

- The design and implementation of modern programming languages that became widespread such as C++, Java, which rely heavily on the concept of theory of automata's context-free languages. Context-free grammars are used to document the language's syntax, and serve as the parsing techniques for all modern language compilers.
- Lexical analysis in compilers. Finite automata identifies keywords, operators, identifiers in input source code.
- Pattern recognition with regular expressions. Finite automata used for searching and matching patterns in text files.
- Text editors. Used to find and replacing patterns in large text files.
- Spelling checkers. Used to verify words in spelling applications
- Decision making and learning. Can be used to help in decision making, learning processes. [8]

References:

[1]. From the Internet:
https://www.bing.com/search?FORM=ARPSEC&PC=ARPL&PTAG=52130265&q=computing%20theoriesdownloaded 12/18/2025 at 12:33pm
[2]. Michael Sipser, "Introduction to the Theory of Computation", 3e, Cengage Learning, 20 Channel Center Street Boston, MA 02210, ISBN-13: 978-1-133-18779-0. Pp: 1-3
[3]. Anil Maheshwari, Michiel Smid, "Introduction to Theory of Computation", School of Computer Science, Carleton University, Ottawa, Canada, March 23, 2017. Pp. 1-2.
[4]. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, "Introduction to Automata Theory, Laguages, and Computation", 2e, Addison-Wesley, 2001, ISBN: 0-201-44124-1. Pp: 1-2.
[5]. From the Internet:https://www.geeksforgeeks.org/theory-of-computation/introduction-of-finite-automata/ downloaded 12/21/2025 at 3:19pm
[6]. John C. Martin, "Introduction to Language and the Theory of Computation", 4e, Mc Graw Hill, New York, NY 10020, 2011, pp: 45-48
[7]. From the Internet:
https://www.bing.com/search?FORM=ARPSEC&PC=ARPL&PTAG=52130265&q=lexical%20analysis downloaded 12/24/2025 12:03pm
[8]. From the Internet: https://www.geeksforgeeks.org/theory-of-computation/applications-of-various-automata/ downloaded 12/24/2025 at 5:14pm