

AUTOMATON-AN ABSTRACT COMPUTING DEVICES & ITS RECENT TRENDS

AKELLA SUBHADRA

M.Tech, M.Phil, ASSOCIATE PROFESSOR IN THE DEPARTMENT OF CSE
BVCITS, BATLAPALEM, AMALAPURAM

ABSTRACT

The theory of formal languages arose in the 50's when computer scientists were trying to use computers to translate one language to the another. Although not much was achieved at that time in translation of natural languages, it paved the way for describing computer languages such as ALGOL 60 later. In the mid of thirties Turing machine was introduced to develop the theory of computability. It turned out that Turing machines were the most general devices for accepting formal languages. Now the theory of formal languages, automata and computation which have emerged respectively as mathematical models of programming languages, computers and capability of a computer have wide range of applications in compiling techniques, robotics, artificial intelligence and knowledge engineering. Automata theory is a branch of computer science that deals with designing abstract self propelled computing devices that follow a predetermined sequence of operations automatically. Formal languages and automata theory is a complicated subject historically only tackled by well trained and experienced experts however as more and more people become aware of this an increasing number of people need to understand the basics of automata theory, languages and computation. This survey provides a comparative overview of general theory of automata, the basics of formal languages, pushdown automata and its relation with context free languages, linear bounded automata and Turing machines. Furthermore we explicitly describe several recent trends of automaton like Grammar systems Distributed automata, Regulated rewriting, A. Lindenmayer systems, membrane systems a molecular model for computing and DNA computing, natural computing, Distributed computing, Contextual grammar

KEYWORDS: Membrane computing, PDA, FSM, TM, Lindenmayer systems

1. INTRODUCTION

Automata theory is an exciting theoretical branch of computer science. It established its roots during the 20th century, as mathematician began developing both theoretically and literally machines which imitated certain features of man. Completing calculations more quickly and reliably. The word automaton itself closely related to the word "automaton" denotes automatic process carrying out the production of specific processes. Simply stated automata theory deals with the logic of computation with respect to simple machines referred to as automata. Through automata computer scientists are able to understand how machines compute functions and solve problems and more importantly what it means for a function to be defined as computable or for a question to be described as decidable. Automata are abstract models of machines that perform computations on an input by moving through a series of states or configurations. At each state of the computation a transition function determines the next configuration on the basis of finite portion of the present configuration. As a result once the computation reaches an accepting configuration it accepts that input. The most general and powerful automata is the Turing machine.

2. WHAT IS AUTOMATA THEORY?

Automaton = an abstract computing device Note: A "device" need not even be a physical hardware!

A moving mechanical device made in imitation of a human being.

A machine that performs a function according to a predetermined set of coded instructions.

An automaton is defined as a system where energy, materials and information are transformed, transmitted and used for performing some functions without direct participation of a man. Examples are Automatic machine tools, automatic packing machines, and automatic photo printing machines

3. OBJECTIVES

The major objective of the automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems in which signals are sampled periodically. The behavior of these discrete systems is determined by the way that the system is constructed from storage and combinational elements. Characteristics of such machines include

Inputs: Assumed to be sequence of symbols selected from a finite set I of input signals namely set I is the set $\{X_1, X_2, X_3, \dots, X_K\}$ where K is the number of inputs.

Outputs: Sequences of symbols selected from a finite set Z namely set Z is the set $\{y_1, y_2, y_3, \dots, y_m\}$ where m is the number of outputs.

States: Finite set Q whose definition depends on the type automaton.

4. WHY STUDY AUTOMATA THEORY?

Finite automata are a useful model for many important kinds of software and hardware:

1. Software for designing and checking the behavior of digital circuits
2. The lexical analyzer of a typical compiler, that is, the compiler component that breaks the input text into logical units
3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases or other patterns
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols of protocols for Secure exchange information

5. MAJOR FAMILIES OF AUTOMATON

The families of automata below can be interpreted in a hierarchical form, where the finite state machine is the simplest automata and the turing machine is the most complex

1. Finite state Automata
2. Pushdown Automata
3. Linear bounded Automata
4. Turing machine

5.1 FINITE STATE AUTOMATA

It was introduced first by two neuro-psychologist Warren S. McCullough and Walter Pitts 1943 as a model for human brain! Applications in digital circuit/protocol verification, compilers, pattern recognition, etc.

The exciting history of how finite automata become a branch of computer science illustrates its wide range of applications. The first people to consider the concept of a finite state machine included a team of biologists, psychologists, mathematicians, engineers and some of the first computer scientists. They all shared a common interest: to model the human thought process, whether in the brain or in a computer. An automaton in which the state set Q contains only a finite number of elements is called a finite state machine (FSM). FSMS are abstract machines consisting of a set of states (set Q), set of input events (set I) and a set of output events (set Z) and a state transition function. The state transition function takes the current state and an input event and returns the new set of output events and the next state.

Finite state machines are ideal computation models for a small amount of memory and do not maintain memory. This mathematical model of a machine can only reach a finite number of states and transition between these states.

Finite Automaton can be classified into two types:

- A) Deterministic Finite Automaton (DFA)
- B) Non-deterministic Finite Automaton (NFA / NFA)

5.1 A) Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

Formal Definition of a DFA

A DFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states.
- Σ is a finite set of symbols, called the **alphabet** of the automaton.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram**.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

5.1 B) Non-deterministic Finite Automaton (NFA)

In NFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NFA

An NFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- **Q** is a finite set of states.
- **Σ** is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the transition function where $\delta: Q \times \{\Sigma \cup \epsilon\} \rightarrow 2Q$
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

There exist several types of finite state machines which can be divided into three main categories

Acceptors, Classifiers, and Transducers

Acceptor (Recognizer) An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier A classifier has more than two final states and it gives a single output when it terminates.

Transducer An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types: Mealy Machine The output depends only on the current state. Moore Machine The output depends both on the current input as well as the current state.

5.2 PUSH DOWN AUTOMATA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is: "**Finite state machine**" + "**a stack**"

A pushdown automaton has three components: an input tape, a control unit, and a stack with infinite size. The stack head scans the top symbol of the stack.

A stack does two operations:

Push: a new symbol is added at the top. , **Pop**: the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

A PDA can be formally described as a 7-tuple $(Q, \Sigma, S, \delta, q_0, I, F)$:

- **Q** is the finite number of states
- **Σ** is input alphabet
- **S** is stack symbols
- **δ** is the transition function: $Q \times (\Sigma \cup \{\epsilon\}) \times S \times Q \times S^*$
- **q_0** is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol ($I \in S$)
- **F** is a set of accepting states ($F \subseteq Q$)

PARSING AND PDA: Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types:

Top-Down Parser: Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.

Bottom-Up Parser: Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions:

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions:

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.

- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

5.3 LINEAR BOUNDED AUTOMATA

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

Here, Memory information $\leq c \times$ Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

Formal Definition of linear bounded automata:

A linear bounded automaton can be defined as an 8-tuple $(Q, X, \Sigma, q_0, ML, MR, \delta, F)$ where:

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **q_0** is the initial state
- **ML** is the left end marker
- **MR** is the right end marker where $MR \neq ML$
- **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- **F** is the set of final states

A deterministic linear bounded automaton is always context-sensitive and the linear bounded automaton with empty language is undecidable

5.4 TURING MACHINE

Alan Turing (1912-1954) (A pioneer of automata theory) Father of Modern Computer Science, English mathematician Studied abstract machines called *Turing machines* even before computers existed

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing. It is an Abstract model of computation Turing machines are widely accepted as a synonyms for algorithmic computability (Church-Turing thesis)–Using these conceptual machines Turing showed that first-order logic validity problem is non-computable. I.e. there exists some problems for which you can never write a program no matter how hard you try!

Formal Definition of a Turing machine

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where:

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **δ** is a transition function; $\delta : Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}\}$
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states

5.4.1 NON-DETERMINISTIC TURING MACHINE

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration. An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a Decider and if for some input, all branches are rejected, the input is also rejected.

Formal Definition of a Non deterministic Turing machine

A non-deterministic Turing machine can be formally defined as a 7-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where:

- **Q** is a finite set of states

- X is the tape alphabet
- Σ is the input alphabet
- δ is a transition function;
 $\delta : Q \times X \rightarrow P(Q \times X \times \{\text{Left_shift}, \text{Right_shift}\})$.
- q_0 is the initial state
- B is the blank symbol
- F is the set of final states

5.4.2 DIFFERENT VARIATIONS OF TURING MACHINES:

1. Turing machines with two way infinite tapes
2. Multiple Turing machines
3. Multi head Turing machines
4. Non deterministic Turing machines
5. Turing machines with two dimensional tapes

6 GRAMMAR

Grammars denote syntactical rules for conversation in natural languages. Linguistics have attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc. The theory of formal languages finds its applicability extensively in the fields of Computer Science. **Noam Chomsky** gave a mathematical model of grammar in 1956 which is effective for writing computer languages.

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where

- **N** or **VN** is a set of Non-terminal symbols
- **T** is a set of Terminal symbols
- **S** is the Start symbol, $S \in N$
- **P** is Production rules for Terminals and Non-terminals

Grammar G1: ($\{S, A, B\}$, $\{a, b\}$, S, $\{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$)

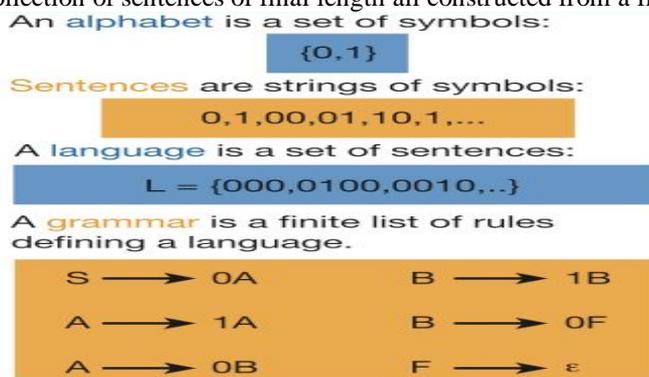
Here, **S**, **A**, and **B** are Non-terminal symbols;

a and **b** are Terminal symbols

S is the Start symbol, $S \in N$

Productions, **P** : $S \rightarrow AB, A \rightarrow a, B \rightarrow b$

Language: A language is a collection of sentences of final length all constructed from a finite alphabet of symbols



Chomsky Classification Of Grammars: According to Noam Chomsky there are four types of grammars, Type 0, Type 1, Type 2, and Type 3

Grammar Type	Grammar Acceptor	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

7 RECENT TRENDS

7.1 L System –Computer imagery

L systems were defined by A. Lindenmayer in an attempt to describe the development of multicellular organisms. In the study of developmental biology, the important changes that take place in cells and tissues during development are considered. L systems provide a framework within which these aspects of development can be expressed in a formal manner.

L systems also provide a way to generate interesting classes of pictures by generating strings and interpreting the symbols of the string as the moves of a cursor.

L system –biological motivation

From the formal language theory point of view, *L* systems differ from the Chomsky grammars in three ways.

- Parallel rewriting of symbols is done at every step. This is the major difference.
- There is no distinction between non terminals and terminals
- Starting point is a string called the axiom.

L system example:

Consider the following DPOL system $\pi_2 = (\Sigma; 4; P)$

where $\Sigma = \{0; 1; 2; \dots; 9; (,)\}$

P has rules

0 → 10; 1 → 32; 2 → 3(4);

3 → 3; 4 → 56; 5 → 37;

6 → 58; 7 → 3(9); 8 → 50;

9 → 39 (!→(,) →)

Ten steps in the derivation are given below

1 4, 2 56, 3 3758

4 33(9)3750,

5 33(39)33(9)3710

6 33(339)33(39)33(9)3210

7 33(3339)33(339)33(39)33(4)3210

8 33(33339)33(3339)33(339)33(56)33(4)3210

9 33(333339)33(33339)33(3339)33(3758)33(56)33(4)3210

10 33(3333339)33(333339)33(33339)33(33(9) 3750)33(3758)33(56)33(4)3210

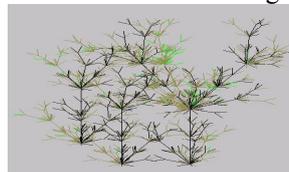
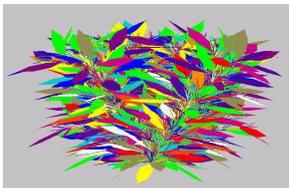
Two dimensional generations of patterns: Here, the description of the string is captured as a string of symbols. An *L*-system is used to generate this string. This string of symbols is viewed as commands controlling a LOGO-like turtle. The basic commands used are move forward, make right turn, make left turn etc. Line segments are drawn in various directions specified by the symbols to generate the straight line pattern. Since most of the patterns have smooth curves, the positions after each move of the turtle are taken as control points for *B*-spline interpolation. We see that this approach is simple and concise.

L system and computer imagery: generation of plant structure

F+ F+ F+ F

F ↔ draw a line of unit length

+ ↔ turn anti clock wise through 90 degrees



APPLICATIONS:

L systems are used for a number of applications in computer imagery. It is used in the generation of fractals, plants, and for object modeling in three dimensions. Applications of *L*-systems can be extended to reproduce traditional art and to compose music.

7.2 NATURAL COMPUTING

Natural computing is the computational version of the process of extracting ideas from nature to develop computational systems, or using natural materials

(1) Computing inspired by nature: it makes use of nature as inspiration for the development of problem solving techniques. The main idea of this branch is to develop computational tools (algorithms) by taking inspiration from nature for the solution of complex problems.

(2) The simulation and emulation of nature by means of computing: it is basically a synthetic process aimed at creating patterns, forms, behaviors, and organisms that (do not necessarily) resemble 'life-as-we-know-it'. Its products can be used to mimic various natural phenomena, thus increasing our understanding of nature and insights about computer models.

(3) Computing with natural materials: it corresponds to the use of novel natural materials to perform computation, thus constituting a true novel computing paradigm that comes to substitute or supplement the current silicon-based computers.

Therefore, natural computing can be defined as the field of research that, based on or inspired by nature, allows the development of new computational tools (in software, hardware or 'wetware') for problem solving, leads to the synthesis of natural patterns, behaviors, and organisms, and may result in the design of novel computing systems that use natural media to computer. Natural computing is thus a field of research that testimonies against the specialization of disciplines in science. It shows, with its three main areas of investigation, that knowledge from various fields of research are necessary for a better understanding of life, for the study and simulation of natural systems and processes, and for the proposal of novel computing paradigms. Physicists, chemists, engineers, biologists, computer scientists, among others, all have to act together or at least share ideas and knowledge in order to make natural computing feasible.

Most of the computational approaches natural computing deals with are based on highly simplified versions of the mechanisms and processes present in the corresponding natural phenomena. The reasons for such simplifications and abstractions are manifold. First of all, most simplifications are necessary to make the computation with a large number of entities tractable. Also, it can be advantageous to highlight the minimal features necessary to enable some particular aspects of a system to be reproduced and to observe some emergent properties.

7.3 DNA COMPUTING

The 'machine' language that describes the objects and processes of living systems contains four letters {A,C,T,G}, and the 'text' that describes a person has a great number of characters. These form the basis of the DNA molecules that contain the genetic information of all living beings. DNA is also the main component of DNA computing.

DNA computing is one particular component of a large field called molecular computing that can be broadly defined as the use of (bio)molecules and bio molecular operations to solve problems and to perform computation. It was introduced by L. Adleman in 1994 when he solved an NP-complete problem using DNA molecules and bio molecular techniques for manipulating DNA. The basic idea is that it is possible to apply operations to a set of (bio)molecules, resulting in interesting and practical performances. It is possible to divide the DNA computing models in two major classes. A first class, commonly referred to as filtering models, which includes models based on operations that are successfully implemented in the laboratory. A second class composed of the so-called formal models, such as the splicing systems and the sticker systems whose properties are much easier to study, but only the first steps have been taken toward their practical implementation.

7.4 REGULATED REWRITING:

In a given grammar, re-writing can take place at a step of a derivation by the usage of any applicable rule in any desired place. That is if A is a non terminal occurring in any sentential form say $\alpha\beta A$ the rules being $A \rightarrow \gamma$ and $A \rightarrow \delta$ then any of these two rules are applicable for the occurrence of A in $\alpha\beta A$. Hence, one encounters non determinism in its application. One way of naturally restricting the nondeterminism is by regulating devices, which can select only certain derivations as correct in such a way that the obtained language has certain useful properties. For example, a very simple and natural control on regular rules may yield a non regular language.

While defining the four types of grammars, we put restrictions in the form of production rules to go from type 0 to type 1, then to type 2 and type 3. In this chapter we put restrictions on the manner of applying the rules and study the effect. There are several methods to control re-writing, some of the standard control strategies are as follows

7.4.1. MATRIX GRAMMER: A matrix grammar is a quadruple $G = (N, T, P, S)$ where N , T and S are as in any Chomsky grammar. P is a finite set of sequences of the form: $m = [\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \alpha_3 \rightarrow \beta_3, \dots, \alpha_n \rightarrow \beta_n]$ where $n \geq 1$,

Where $\alpha_i \in (NUT)^+$, $\beta_i \in (NUT)^*$, $1 \leq i \leq n$, m is a member of p and a' matrix 'of p

G is a matrix grammar of type I where $I \in \{0, 1, 2, 3\}$, if and only if the grammar $G_m = (N, T, m, S)$ is of type i for every $m \in P$

7.4.2. PROGRAMMED GRAMMER: A Programmed Grammar is a 4-tuple, where N , T and S are as in any Chomsky grammar. Let r be a collection of re-writing rules over NUT , $lab(R)$ being the labels of R . σ, Φ are mappings from $lab(R)$ to $2^{lab(R)}$ $P = \{(r, \sigma(r), \Phi(r)) / r \in R\}$

Here, G is said to be type i , or C -free if the rules in R are all type i , where $i = 0, 1, 2, 3$ or C -free respectively

7.4.3. RANDOM CONTEXT GRAMMAR: A Random context grammar has two sets of non terminals X, Y where the set X is called the permitting context and Y is called the forbidding context of a rule. $x \rightarrow y$

7.4.4. TIME VARYING GRAMMAR: Given a grammar G , one can think of applying a set of rules only for a particular period. That is, the entire set of P is not available at any step of a derivation. Only a subset of P is available at any time 't' or at any i-th step of a derivation.

7.4.5. REGULAR CONTROL GRAMMERS: Let G be a grammar with production set P and $lab(P)$ be the labels of productions of P . To each derivation D , according to G , there corresponds a string over $lab(P)$ (the so called control string). Let C be a language over $lab(P)$. We define a language L generated by a grammar G such that every string of L has a derivation D with a control string from C . Such a language is said to be a controlled language.

7.4.6. INDIAN PARALLEL GRAMMARS: In the definition of matrix, programmed, time-varying, regular control, and random context grammars, only one rule is applied at any step of derivation.

7.5 MEMBRANE COMPUTING:

New field of research, motivated by the way nature computes at the cellular level, introduced by Prof. Gh. Păun. It is also called as **P systems**. A class of distributed parallel computing devices of biochemical type. Membrane computing is a branch of natural computing which investigates computing models abstracted from the structure and functioning of living cells and from their interactions in tissues or higher order biological structures. A membrane system is a distributed computing model processing multi sets of objects either in the compartments of a cell-like hierarchical arrangement of membranes (hence a structure of compartments which corresponds to a rooted tree), or in a tissue-like structure consisting of cells placed in the nodes of an arbitrary graph. First, there are several essential features genuinely relevant to membrane

Membrane computing abstracts computing models from the architecture and the functioning of living cells, as well as from the organization of cells in tissues, organs (brain included) or other higher order structures such as colonies of cells (e.g., bacteria). The initial goal was to learn from cell biology something possibly useful to computer science, and the area quickly developed in this direction. Several classes of computing models { called P systems}, inspired from biological facts or motivated from mathematical or computer science points of view. A number of applications were reported in the last few years in several areas { biology, bio-medicine, linguistics, computer graphics, economics, approximate optimization, cryptography, etc. Several software products for simulating P systems and attempts of implementing P systems on a dedicated hardware were reported; also an attempt towards an implementation in biochemical terms is in progress.

The main ingredients of a P system are (i) the membrane structure, delimiting compartments where (ii) multi sets of objects evolve according to (iii) (reaction) rules of a bio-chemical inspiration. The rules can process both objects and membranes. Thus, membrane computing can be defined as a framework for devising cell-like or tissue-like computing models which process multi sets in compartments defined by means of membranes. These models are (in general) distributed and parallel. When a P system is considered as a computing device, hence it is investigated in terms of (theoretical) computer science, the main issues studied concern the computing power (in comparison with standard models from computability theory, especially Turing machines/Chomsky grammars and their restrictions) and the computing efficiency (the possibility of using parallelism for solving computationally hard problems in a feasible time). Computationally and mathematically oriented ways of using the rules and of defining the result of a computation are considered in this case. When a P system is constructed as a model of a bio-chemical process, it is examined in terms of dynamical systems, with the evolution in time being the issue of interest, not a specific output. From a theoretical point of view, P systems are both powerful (most classes are Turing complete, even when using ingredients of a reduced complexity { a small number of membranes, rules of simple forms, ways of controlling the use of rules directly inspired from biology are sufficient for generating/accepting all sets of numbers or languages generated by Turing machines) and efficient (many classes of P systems, especially those with enhanced parallelism, can solve computationally hard problems { typically NP-complete problems, but also harder problems, e.g., PSPACE-complete problems { in feasible time { typically polynomial). Then, as a modeling framework, membrane computing is rather adequate for handling discrete (biological) processes, having many attractive features: easy understandability, scalability and programmability, inherent compartmentalization and non-linearity, etc.

APPLICATIONS:

1. Distribution (with important issues related to system-part interaction and emergent behavior nonlinearly resulting from the composition of local behaviors),
2. Discrete mathematics (continuous mathematics, especially systems of differential equations, has a glorious history of applications in many disciplines, such as astronomy, physics, and meteorology, but has failed to prove adequate for linguistics, and cannot cover more than local processes in biology because of the complexity of the processes and, in many cases, because of the imprecise character of the processes; a basic question is whether the biological reality is of continuous or discrete nature, as languages proved to be, with the latter ruling out the usefulness of many tools from continuous mathematics),
3. Algorithmicity: P systems are computability models of the same type as Turing machines or other classic representations of algorithms, and, as a consequence, they can be easily simulated on computers),
4. Scalability/extensibility (this is one of the main difficulties of using differential equations in biology),

5. Transparency (multi set rewriting rules are nothing other than reaction equations as customarily used in chemistry and biochemistry, without any \mysterious" notation or \mysterious" behavior),
6. Massive parallelism (a dream of computer science, a commonplace in biology),
7. Nondeterminism (let us compare the \program" of a P system, i.e., a set of rules localized in certain regions and without any imposed ordering, with the rigid sequences of instructions of programs written in typical programming languages),
8. Communication (with the astonishing and still not completely understood way in which life is coordinating the multitude of processes taking place in cells, tissues, organs, and organisms, in contrast with the costly way of coordinating/synchronizing computations in parallel electronic computing architectures, where the communication time becomes prohibitive with the increase in the number of processors).

8 CONCLUSIONS

The simplest automata used for computation is finite automaton. It can compute only primitive functions therefore it is not an adequate computational model. Imagine a modern CPU, every bit in a machine can only be in two states 0 or 1. Therefore there are a finite number of possible states in addition, when considering the parts of a computer a CPU interacts with, there are a finite number of possible inputs from the computer's mouse, keyboard, hard disk, different slot cards etc. As a result one can conclude that a CPU can be modeled as a finite state machine. Now consider a computer, although every bit in a machine can only be in two different states (0 or 1) there are an infinite number of interactions with in the computer as a whole. It becomes exceeding difficult to model the working of a computer within the constraints of a finite state machine. However higher level infinite and more powerful automata would be capable of carrying out this task. The Turing machine can be thought of as a finite automaton or control unit equipped with an infinite storage (memory). Its memory consists of an infinite number of one dimensional array of cells. Turing machine is essentially an abstract model of modern day computer execution and storage, developed in order to provide a precise mathematical definition of an algorithm or mechanical procedure. Hence we conclude that an automaton is an abstract device to the modern purpose computer and its applications are grown rapidly in various fields. ranging from verification to XML processing and file comparison

REFERENCES

- [1] Introduction to theory of computation – Sipser
- [2] Elaine Rich (2008). Automata, Computability and Complexity: Theory and Applications. Pearson. ISBN0-13-228806
- [3] Introduction to Automata Theory, Languages And Computation "by John E. Hopcroft, Motwani, and Ullman
- [4] Anderson, James A. (2006). Automata theory with modern applications. With contributions by Tom Head. Cambridge: Cambridge University Press. ISBN 0-521-61324-8. Zbl 1127.68049.
- [5] Sakarovitch, Jacques (2009). Elements of automata theory. Translated from the French by Reuben Thomas. Cambridge University Press. ISBN 978-0-521-84425-3. Zbl 1188.68177.
- [6] Igor Aleksander, F.Keith Hanna (1975). Automata Theory: An Engineering Approach. New York: Crane Russak. ISBN 0-8448-0657-9.
- [7] An introduction to Finite automata and their connection to Logic-Howard Straubing and Pascal Weil
- [8] Automata & Computability by Dexter Kozen
- [9] Elements of automata theory by Jacques Sakarovitch, Cambridge university press, 2009
- [10] Problem solving in Automata, Languages, And complexity by Du-Ko
- [11] Finite automata Behavior and synthesis by B.A. Trakhtenbort and Ya.M. Barzdin
- [12] Introduction to languages and The Theory of Computation-John C. Martin
- [13] Theory of computation by H. Lewies and C. Papadimitriou
- [14] Formal languages and automata theory by A.A. Puntambekar
- [15] Membrane computing – An introduction by Hendrik Jan Hoogeboom, Gheorghe Paun-2007
- [16] Handbook of natural computing by Grzegorz Rozenberg
- [17] Lindenmayer systems-Impacts on Theoretical computer science by Grzegorz Rozenberg, Arto Salomaa

AUTHOR'S PROFILE:



Ms AKELLA.SUBHADRA born in Amalapuram, AP State, India, Received Master of Computer Applications from Andhra University, Vizag. Received Master of Technology in Computer Science from Nagarjuna University-Guntur and Received Master of Philosophy in Computer Science from Periyar University, Salem. She is presently Associate professor in Bonam Venkata Chalamayya Institute of Technology & Science (BVCITS), Batlapalem, Amalapuram. She is having teaching experience of 10 years. Her areas of interests are Formal Languages and Automata Theory, Information Security, Compiler Design, Data Mining, Computer Networks and Design and Analysis of Algorithms. She is member of ISTE and Computer Society of India.