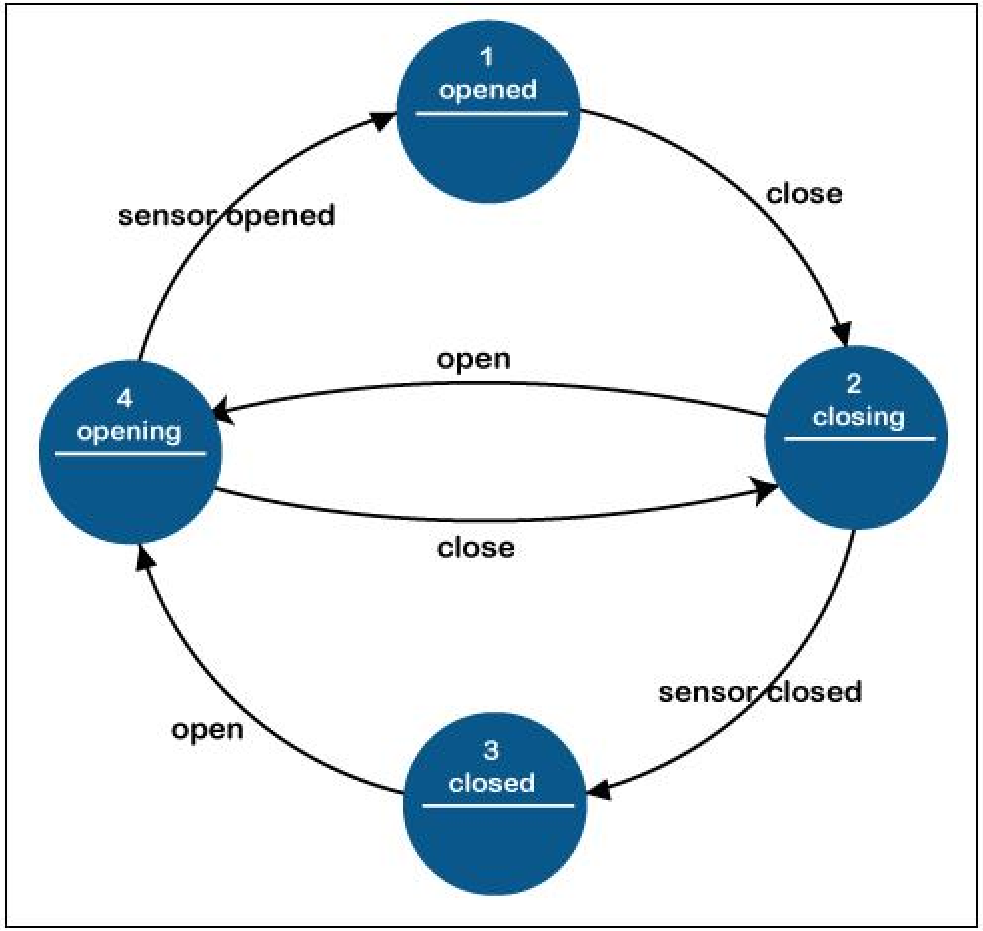
FINITE STATE MACHINE

Introduction

- FSMs, also known as state machines or finite automata, are powerful tools for modeling systems with a finite number of states and transitions between them based on inputs
- scenarios from simple vending machines to complex network protocols
- FSMs find applications in diverse fields such as computer science (e.g., compiler design, protocol specification), engineering (e.g., digital circuit design, robotics), biology (e.g., modeling genetic regulatory networks), and more.
- In 1943, Warren McCulloch and Walter Pitts, two neurophysiologists, were the first to present a formal description of finite automata.

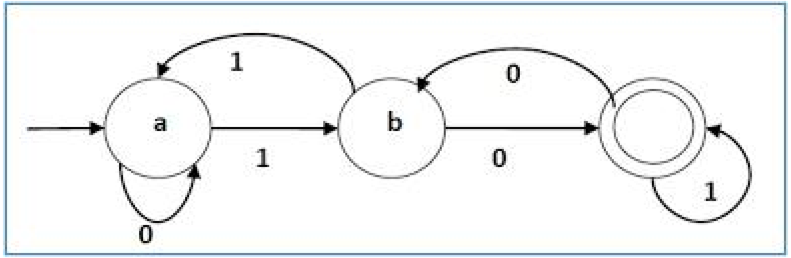
Basic Components

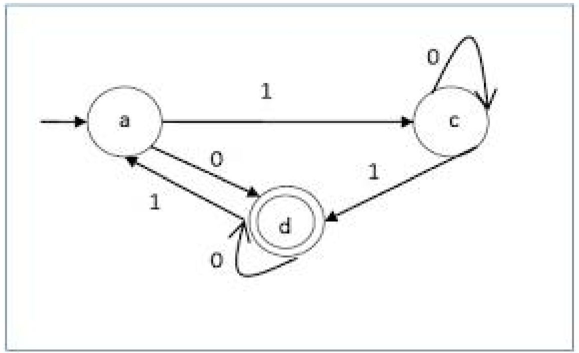
- States: Represent the different configurations or conditions that a system can be in.
- Transitions: Define the movement between states in response to inputs or events.
- Inputs: Stimuli that trigger state transitions.
- Outputs: Responses or actions produced by the system as a result of its current state and input.
- Initial State

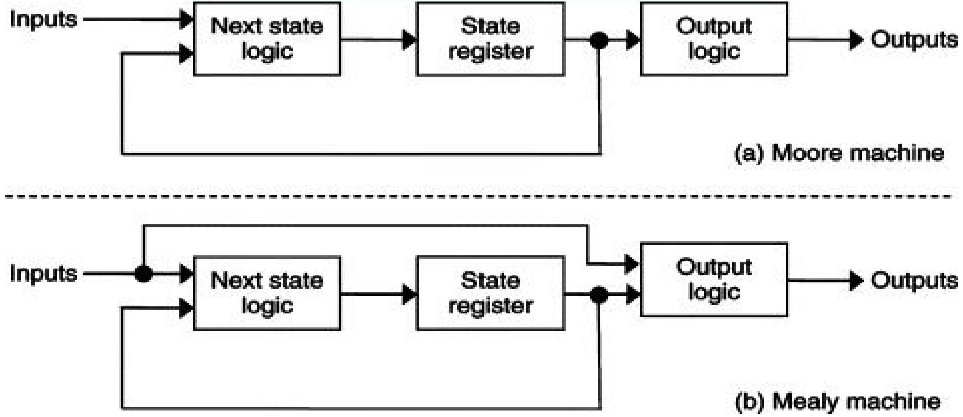


Types of Finite State Machines

- Deterministic Finite State Machines (DFSMs):
  - In a Deterministic FSM, for every state and input, there is exactly one transition leading to a next state. This makes the behavior of the machine predictable and unambiguous.
- Non-Deterministic Finite State Machines (NDFSMs):
  - NDFSMs allow multiple transitions from a state for the same input. The machine may have multiple possible paths to follow based on the input, leading to potential ambiguity in its behavior.
- Mealy Machines:
  - Mealy Machines are FSMs where outputs are produced as soon as a transition occurs. The output depends on both the current state and the input that caused the transition.
- Moore Machines:
  - In contrast to Mealy Machines, Moore Machines produce outputs based solely on the current state, independent of the input that triggered the transition.







(a) Moore machine

(b) Mealy machine

Properties of FSM

- Deterministic and Non-Deterministic Properties:
  - FSMs can exhibit deterministic behavior, where the next state is uniquely determined by the current state and input, or non-deterministic behavior, where there may be multiple possible transitions for the same state and input.
- Reachability and Liveness:
  - Reachability refers to the ability to reach a particular state from the initial state through a sequence of inputs. Liveness indicates whether the system can progress and reach a valid state infinitely often.
- Completeness and Incompleteness:
  - A complete FSM covers all possible input scenarios, ensuring that there is a defined transition for every possible input in every state. Incompleteness implies the presence of undefined or missing transitions.
- Deadlocks and Livelocks:
  - Deadlocks occur when the system reaches a state from which no further transitions are possible, leading to a halt in execution. Livelocks refer to situations where the system keeps transitioning between states without making progress towards a solution.

Representation of FSMs

- State Transition Diagrams (STDs):
  - STDs visually represent FSMs using states as nodes and transitions as directed edges between them. This graphical representation provides an intuitive way to understand the structure and behavior of the system.
- State Transition Tables (STTs):
  - STTs present FSMs in tabular form, listing all possible combinations of states and inputs along with the corresponding next states. This format is useful for concise representation and analysis of complex FSMs.
- Graphical Representation:
  - FSMs can be graphically represented using various diagrams such as state transition diagrams, statecharts, and directed graphs. These visual representations aid in understanding and communicating the behavior of the system effectively.
- Mathematical Formulation:
  - FSMs can also be formally defined using mathematical notation, specifying the set of states, input alphabet, transition function, initial state, and accepting states. This mathematical formulation provides a precise and rigorous description of the FSM.

Finite State Machine Design

- Design Methodologies:
    - FSM design involves systematic methodologies for defining states, inputs, outputs, and transitions based on the requirements of the system. Common methodologies include top-down design, bottom-up design, and state-based design.
    - Ex-Top-down, Bottom-up, State based etc.
- State Minimization Techniques:
    - State minimization aims to reduce the number of states in an FSM while preserving its functionality. Techniques such as state equivalence, state elimination, and state merging are employed to minimize the size of the FSM.
    - Ex-State equivalence, Elimination, Merging, etc.
- State Assignment Techniques:
    - State assignment involves assigning binary codes to states for efficient representation in hardware implementations. Techniques such as one-hot encoding, binary encoding, and gray coding are used to assign unique codes to states.
    - Ex- One Hot encoding, Binary Encoding, Gray coding
- Optimization Strategies:
    - Optimization techniques focus on improving the performance, efficiency, and resource utilization of FSMs. Strategies such as state encoding optimization, transition optimization, and clock gating are employed to optimize the design of FSMs.
    - Ex-State coding Optimization, Transition Optimization, Clock Gating

Applications

- Digital Circuit Design and Verification:
  - FSMs are extensively used in the design and verification of digital circuits, including sequential logic circuits, finite state machines, and control units. They help in modeling and simulating complex digital systems to ensure correct functionality and performance.
- Protocol Specification and Verification:
  - FSMs are employed in the specification and verification of communication protocols, including network protocols, data link protocols, and transport protocols. They facilitate the formal modeling and analysis of protocol behavior to detect errors and ensure protocol correctness.
- Natural Language Processing:
  - FSMs play a vital role in natural language processing tasks such as parsing, tokenization, and part-of-speech tagging. They help in modeling and processing the syntactic structure and semantics of natural language texts to facilitate various language processing applications.
- Compiler Design:
  - FSMs are integral to the design and implementation of compilers, including lexical analyzers, syntax analyzers, and code generators. They aid in parsing and analyzing source code to generate executable machine code efficiently.
- Robotics and Automation:
  - FSMs find applications in robotics and automation systems for modeling and controlling robot behavior, including navigation, manipulation, and decision-making tasks. They help in designing intelligent systems that exhibit complex behavior in response to sensory inputs and environmental conditions.

Beyond FSM

- Hierarchical Finite State Machines:
  - Hierarchical FSMs (HFSMs) are FSMs organized in a hierarchical structure, where states can contain substates and transitions can occur at multiple levels of abstraction. HFSMs provide a modular and scalable approach to modeling complex systems with hierarchical behavior.
- Statecharts:
  - Statecharts are a visual modeling language based on FSMs, extended with features such as hierarchy, concurrency, and orthogonality. Statecharts enable the specification of complex reactive systems with multiple concurrent states and transitions, enhancing the expressiveness and flexibility of FSMs.
- Timed and Probabilistic FSMs:
  - Timed FSMs (TFSMs) incorporate time-dependent behavior, where transitions are triggered based on timing constraints or delays. Probabilistic FSMs (PFSMs) introduce probabilistic transitions, where transition probabilities determine the likelihood of transitioning between states. TFSMs and PFSMs are used to model systems with temporal and stochastic behavior, respectively.
- FSMs in Parallel and Distributed Systems:
  - FSMs are applied in parallel and distributed systems for modeling and coordinating concurrent processes and communication protocols. Distributed FSMs (DFSMs) and Parallel FSMs (PFSMs) enable the specification of distributed and parallel behavior, supporting the design of scalable and efficient distributed systems.

Advantages of FSM

- Simplicity: simple and intuitive model for representing system behavior, making them easy to understand, analyze, and implement.
- Modularity: support modular design, allowing complex systems to be decomposed into smaller, more manageable components or states.
- Formalism: offer a formal mathematical framework for specifying and analyzing system behavior, facilitating rigorous reasoning and verification.
- Efficiency: computationally efficient with constant-time transitions between states, making them suitable for real-time and embedded systems.
- Versatility:can model a wide range of systems and applications across various domains, from digital circuits and protocols to natural language processing and robotics.

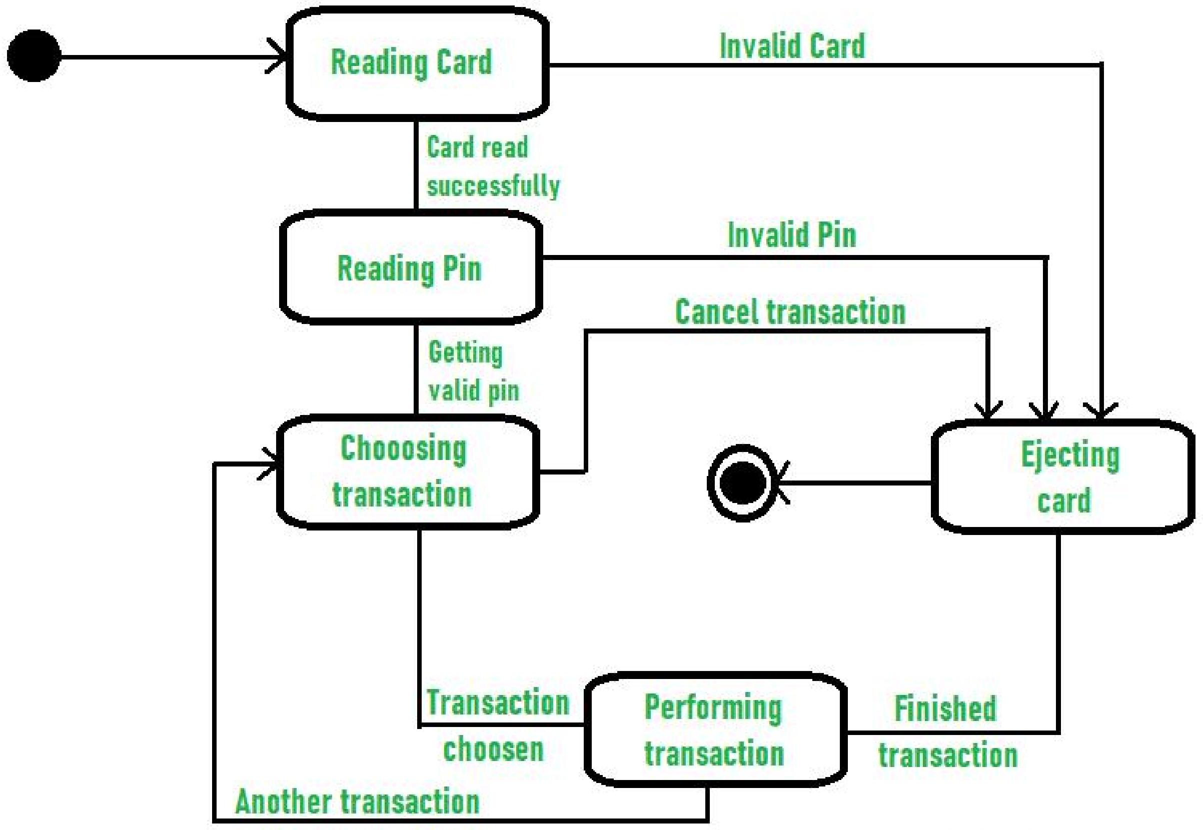
Disadvantages of FSM

- State Explosion: when modeling complex systems with a large number of states, leading to increased design complexity and resource requirements.
- Limited Expressiveness: for modeling dynamic or unbounded behaviors, such as recursive algorithms or infinite sequences.
- Brittleness: when the system requirements change, requiring extensive redesign or modification to accommodate new behaviors or transitions.
- Complexity in Real-world Systems: Real-world systems often exhibit behaviors that are difficult to capture using traditional FSMs, such as concurrency, timing constraints, and probabilistic behavior.
- Maintenance Challenges: can become difficult to maintain and evolve over time, especially as the system grows in complexity or undergoes frequent changes.

State Chart

- Statecharts are a visual modeling language used to represent the behavior of complex systems in the form of boxes and arrows. They extend traditional finite state machines (FSMs) by introducing hierarchical structure, concurrency, and orthogonality, allowing for more expressive and modular system modeling.
- Statecharts provide a powerful tool for modeling and analyzing the behavior of systems in various domains, including software engineering, embedded systems design, control systems, and more.
- Daniel Harel developed statecharts in 1987, which he described as a visual formalism for complex systems.

Example of state chart



State Transition Diagram for ATM System

Components of State Chart

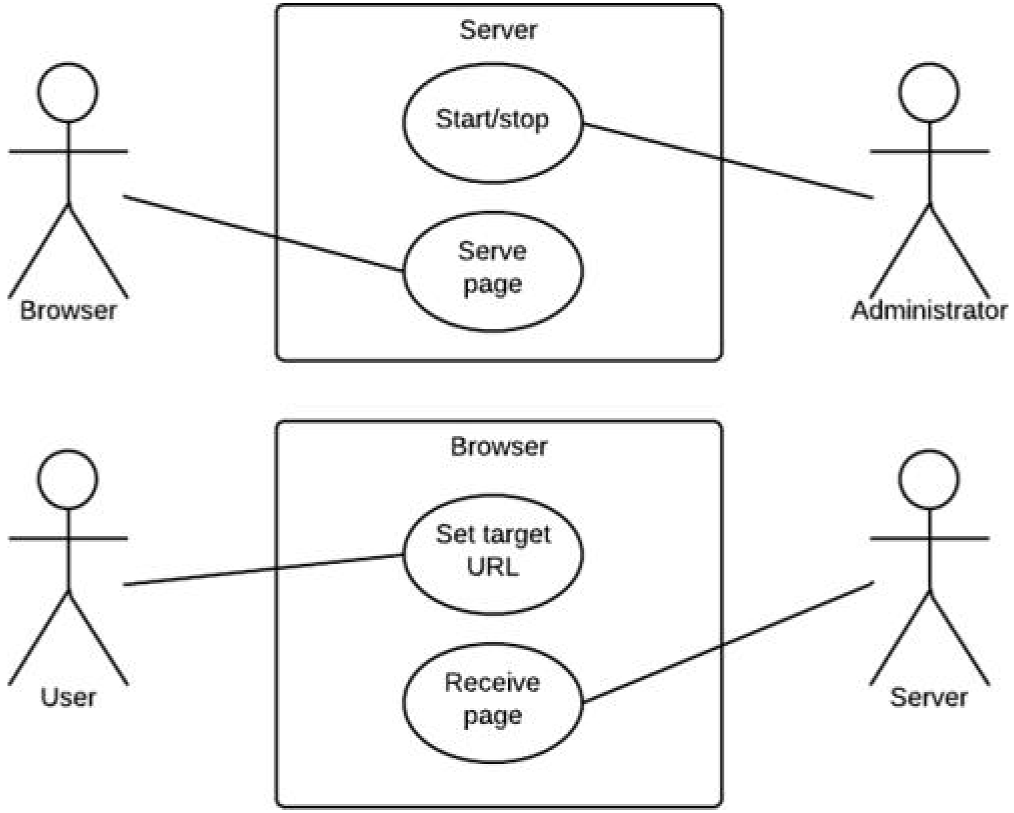
- States: These represent distinct conditions or situations the system can be in, such as "idle," "active," or "error."
    - Superstate
    - Substate
    - Basic state
- Transitions: These are directed connections between states, indicating how the system moves from one state to another. Transitions are triggered by specific events.
- Events: These are external stimuli that cause the system to change its state. Examples might include user input, sensor readings, or timer expirations.
- Actions: These are activities performed by the system. They can occur upon entering or exiting a state, or during a transition. Actions might involve sending data, updating variables, or interacting with the environment.

Types of state chart

- Mealy Machine
- Moore Machine
- Harel Statechart
- UML (Unified Modeling Language): for modeling system behavior
- SDL (Specification and Description Language)

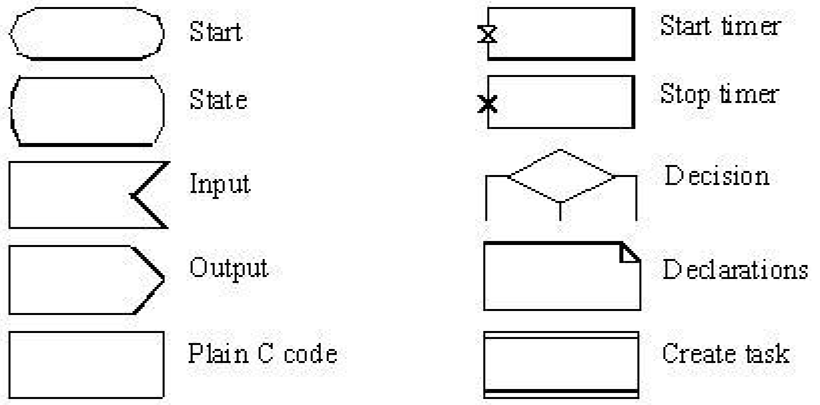
UML

- standardized modeling language used in software engineering to visually represent systems, their components, and their interactions
- serves as a communication tool between stakeholders involved in software development, including developers, designers, architects, and clients



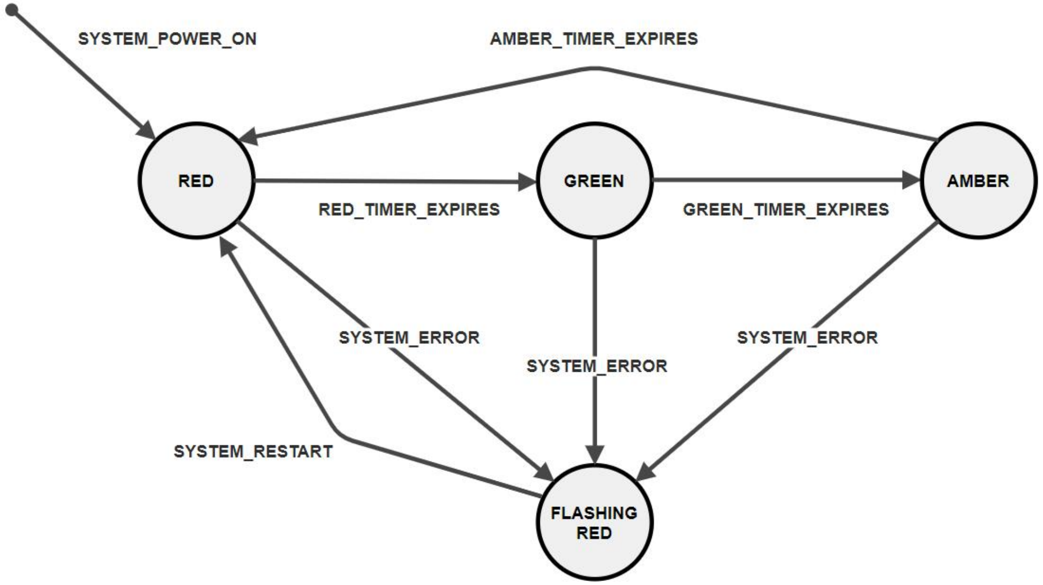
SDL

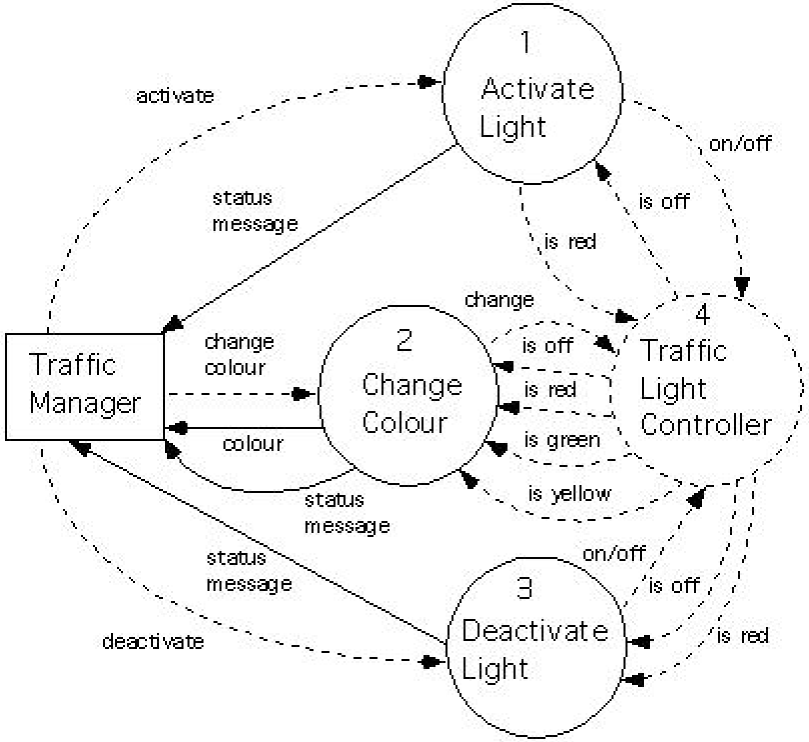
- another formal notation for state charts used in telecommunication and embedded system design. It offers a rich set of features for modeling complex real-time systems and includes functionalities for specifying communication protocols and data exchange between states



Traffic Control System: Traditional State Chart

- States: This refers to the different states that a traffic light can be in, such as "Red", "Yellow", and "Green". Each state represents a specific configuration of the traffic light.
- Transitions: These define the possible changes from one state to another based on certain conditions or events. For example, the transition from "Red" to "Green" might occur when the timer for the green light expires.
- Events: Events trigger transitions between states. These events can be external stimuli like the press of a pedestrian crossing button or internal events like timer expiration.


SYSTEM_POWER_ON
AMBER_TIMER_EXPIRES
RED
GREEN
AMBER
RED_TIMER_EXPIRES
GREEN_TIMER_EXPIRES
SYSTEM_ERROR
SYSTEM_ERROR
SYSTEM_ERROR
SYSTEM_RESTART
FLASHING RED


1
Activate Light
activate
on/off
status message
is off
is red
change
2
Change Colour
4
Traffic Light Controller
Traffic Manager
change colour
is off
is red
is green
is yellow
colour
status message
status message
on/off
is off
is red
3
Deactivate Light
deactivate

- Consider a traffic light intersection with two main roads and a pedestrian crossing. The traffic light for each main road follows a cycle of 60 seconds, consisting of 30 seconds for the green light, 10 seconds for the yellow light, and 20 seconds for the red light. The pedestrian crossing signal activates every 90 seconds for 15 seconds. Calculate the total cycle time for the traffic light intersection, including the pedestrian crossing signal.

- To calculate the total cycle time for the traffic light intersection, we need to consider the time for each phase of the traffic lights and the pedestrian crossing signal.
- Total cycle time = Time for main road traffic lights + Time for pedestrian crossing signal
- Time for main road traffic lights:
- Green light: 30 seconds
- Yellow light: 10 seconds
- Red light: 20 seconds
- Total time for one cycle = 30 + 10 + 20 = 60 seconds
- Time for pedestrian crossing signal:
- Activation time: 15 seconds
- Interval between activations: 90 seconds
- Total time for one cycle = 15 seconds (activation time) + 90 seconds (interval) = 105 seconds
- Total cycle time for the traffic light intersection:
- Total cycle time = 60 seconds (main road traffic lights) + 105 seconds (pedestrian crossing signal) = 165 seconds

- A traffic light intersection has two main roads, North-South (NS) and East-West (EW). The NS road has a traffic flow of 600 vehicles per hour, and the EW road has a traffic flow of 400 vehicles per hour. Each cycle of the traffic light is 2 minutes long. Determine the duration of the green light for each direction to optimize traffic flow.

- To optimize traffic flow, we need to allocate the green light duration based on the traffic flow rates of each direction. We can use a proportional allocation approach where the duration of the green light is proportional to the traffic flow rate.
- Calculations:
- Traffic Flow Rate (NS) = 600 vehicles/hour
- Traffic Flow Rate (EW) = 400 vehicles/hour
- Total Traffic Flow = Traffic Flow Rate (NS) + Traffic Flow Rate (EW) = 600 + 400 = 1000 vehicles/hour
- Green Light Allocation:
- Green Light Duration (NS) = (Traffic Flow Rate (NS) / Total Traffic Flow) * Cycle Duration
- = (600 / 1000) * 120 seconds = 72 seconds
- Green Light Duration (EW) = (Traffic Flow Rate (EW) / Total Traffic Flow) * Cycle Duration
- = (400 / 1000) * 120 seconds = 48 seconds

Hierarchial state chart: Robotic arm control

- High-level (Task): Pick up a cup from the table.
- Mid-level (Behavior): Move the arm towards the cup, grasp it, and lift it.
- Low-level Control: Control individual joints to achieve desired arm movement and gripper force.

State Chart vs Flow Chart

- State Chart performs actions in response to explicit events. In contrast, the flowchart does not need explicit events but rather transitions from node to node in its graph automatically upon completion of activities.
- State chart is used to represent the structure of an organization or system, while a flowchart is used to illustrate the sequence and flow of a process or activity.

- Design a state chart to model the behavior of a robot navigating a maze. The robot can move forward, turn left, or turn right. It starts at the entrance of the maze and aims to reach the exit. The maze has walls, empty spaces, and goal locations. The robot can sense its immediate surroundings (front, left, and right) using sensors.
- The state chart should capture the robot's decision-making process based on sensor data and guide it towards the exit while avoiding collisions.

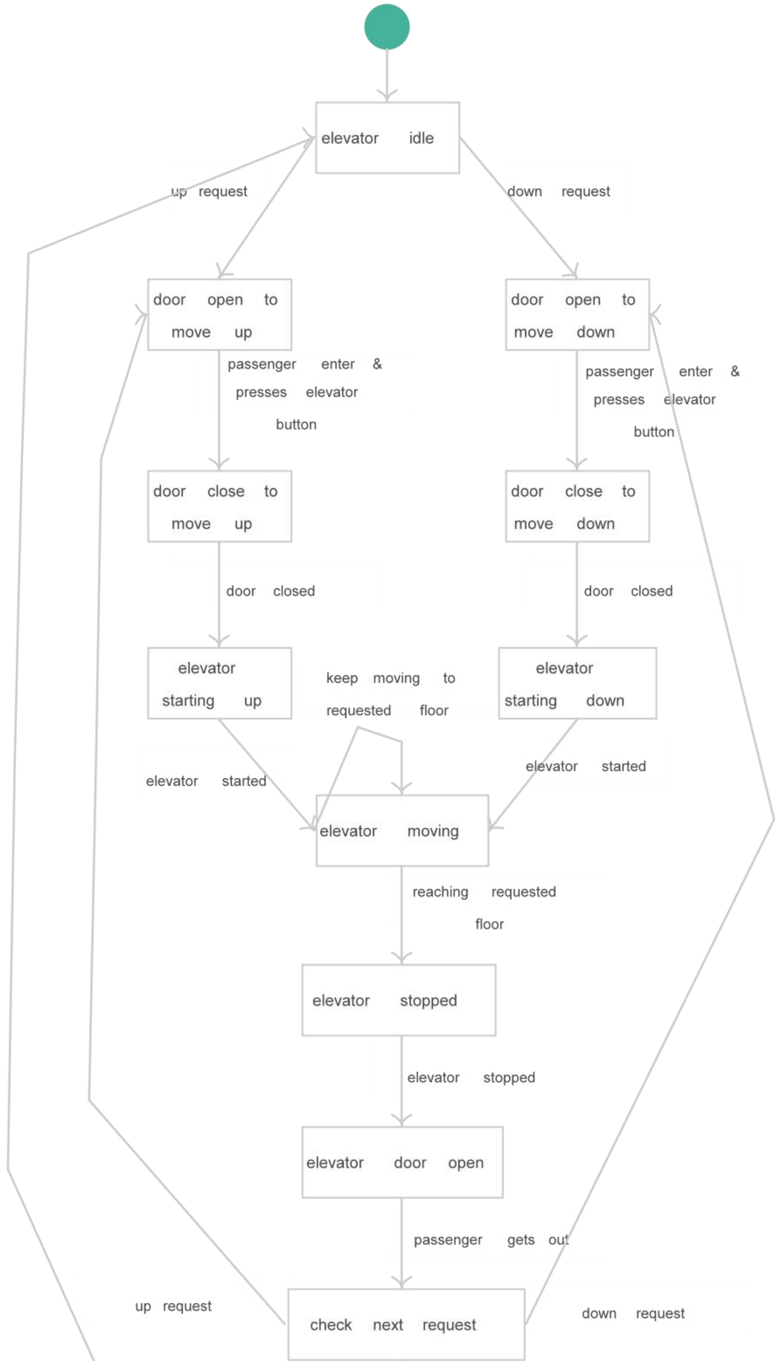
- The state chart includes the following states:
  - Start: The initial state where the robot begins its navigation.
  - Move Forward: The robot moves forward until it encounters a wall or reaches the goal.
  - Turn Left: The robot turns left 90 degrees.
  - Turn Right: The robot turns right 90 degrees.
  - Goal: The robot reaches the exit of the maze and successfully completes the task.
  - Collision: The robot encounters a wall and needs to change its direction.
- Transitions between states are triggered by events based on sensor readings:
  - Start -> Move Forward: The robot starts moving forward.
  - Move Forward -> Move Forward, Turn Left, or Turn Right: Depending on the sensor data (wall in front, left, or right), the robot continues moving, turns left, or turns right, respectively.
  - Turn Left/Right -> Move Forward: After turning, the robot attempts to move forward again.
  - Move Forward -> Goal: If the robot moves forward and reaches the goal location, the navigation is successful.
  - Move Forward -> Collision: If the robot moves forward and encounters a wall, it transitions to the collision state.
  - Collision -> Turn Left or Turn Right: The robot randomly chooses to turn left or right to escape the collision and find a new path.

Orthogonal State Chart

- Multi-Thread Control Systems: programming technique that enables a program to execute multiple threads concurrently, allowing for parallel execution of tasks. Ex- sensor data processing, actuator control, or user interface management, where multiple tasks need to be executed concurrently for efficient system operation.
- orthogonal regions promote modularity by allowing the system to be divided into smaller, independent components. Each orthogonal region encapsulates a specific aspect of the system's functionality, making it easier to understand, develop, and maintain.
- examples of tasks such as data acquisition, filtering, fusion, and interpretation, each running independently within its own orthogonal region.
- reusability reduces development time and effort by leveraging existing components, promoting consistency and standardization across projects.

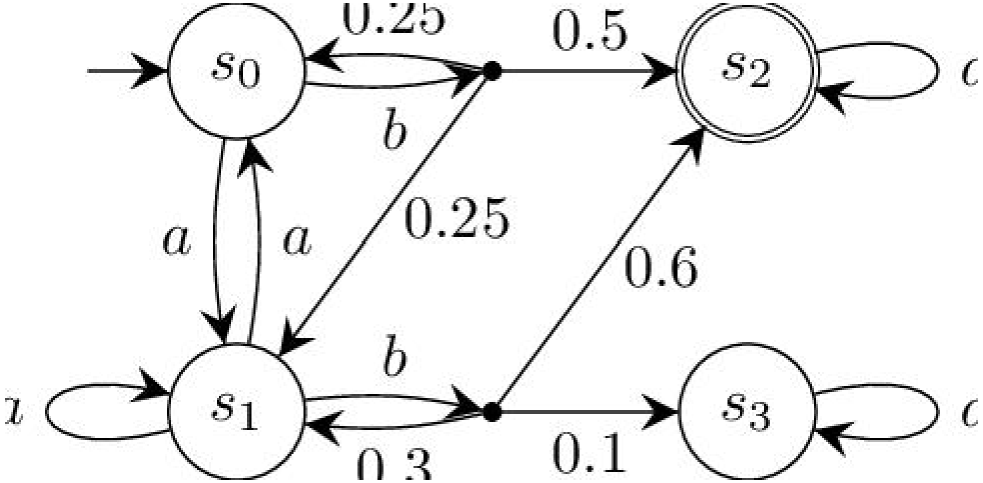
Timed State Chart

- timed statecharts extend traditional statecharts by incorporating time-dependent states and transitions
- in addition to events triggering state transitions, timed statecharts include conditions based on elapsed time or specific time intervals to determine when transitions occur
- ability to model time-critical behaviors accurately, ensuring that the system meets its timing requirements and operates reliably in time-constrained environments
- Ex- Elevator System



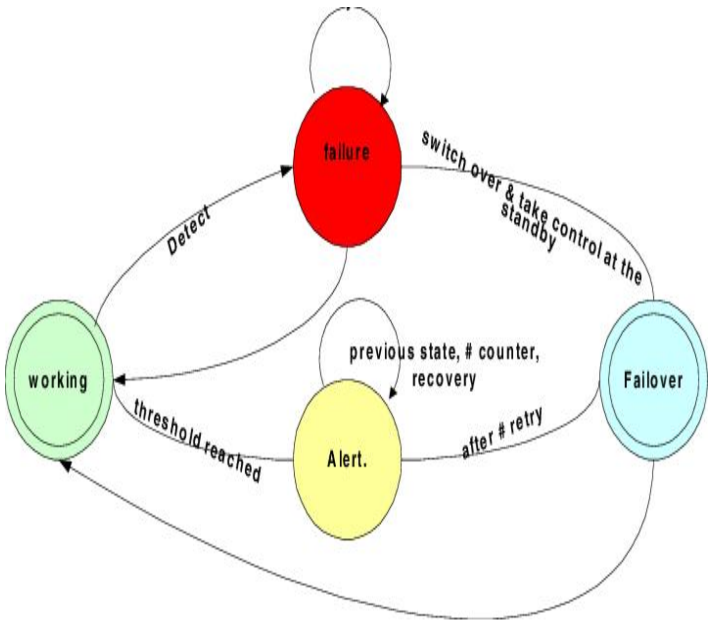
Probabilistic State Chart

- scenarios where task execution times or resource availability are uncertain and subject to variability.
- tasks with varying execution times due to external factors or resources that may become available or unavailable randomly
- workload distribution states, such as task assignment, migration, or balancing



Dynamic State Chart

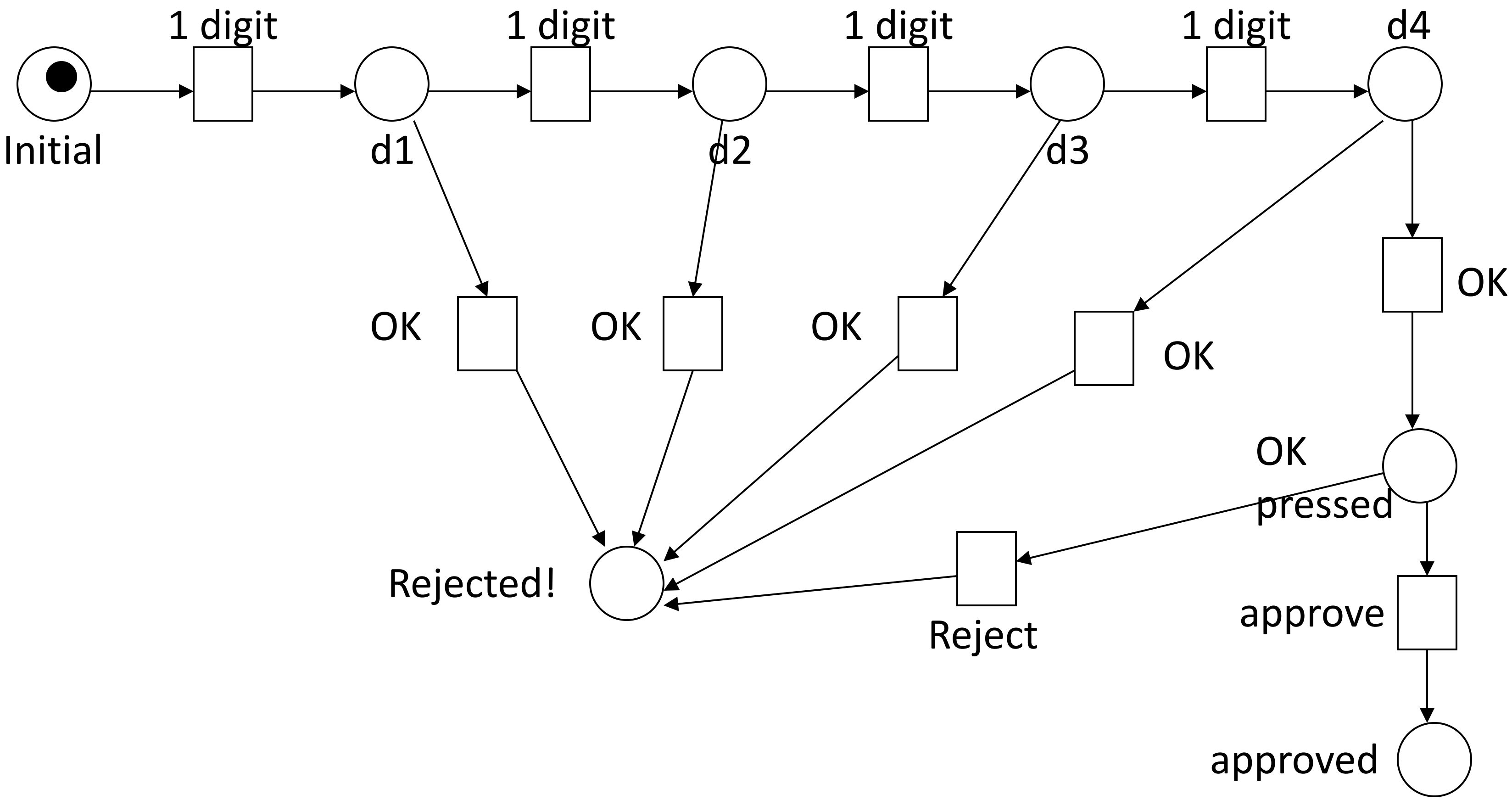
- extend traditional statecharts by incorporating dynamic states and transitions, which allow the system to adapt to changing conditions in real-time
- benefits of utilizing dynamic statechart modeling in self-healing network protocols. includes the ability to dynamically adjust the protocol's behavior to respond to network events or faults, resulting in improved resilience, adaptability, and efficiency in network management and recovery.


failure
switch over & take control at the standby
Detect
working
previous state, # counter, recovery
Failover
threshold reached
Alert.
after # retry

Petrinet

- First introduced by Carl Adam Petri in 1962.
- A diagrammatic tool to model concurrency and synchronization in distributed systems.
- Very similar to State Transition Diagrams.
- Used as a visual communication aid to model the system behaviour.
- Based on strong mathematical foundation.

EFTPOS= Electronic Fund Transfer Point of Sale



Elements of Petrinet

- Places: Represent conditions or states in the system.
- Transitions: Represent events or actions that can occur in the system.
- Arcs: Connect places and transitions, indicating the flow of tokens.
- Tokens: Represent the presence or absence of resources or conditions in places.
- Markings: Describe the distribution of tokens in the Petri Net at a given point in time.

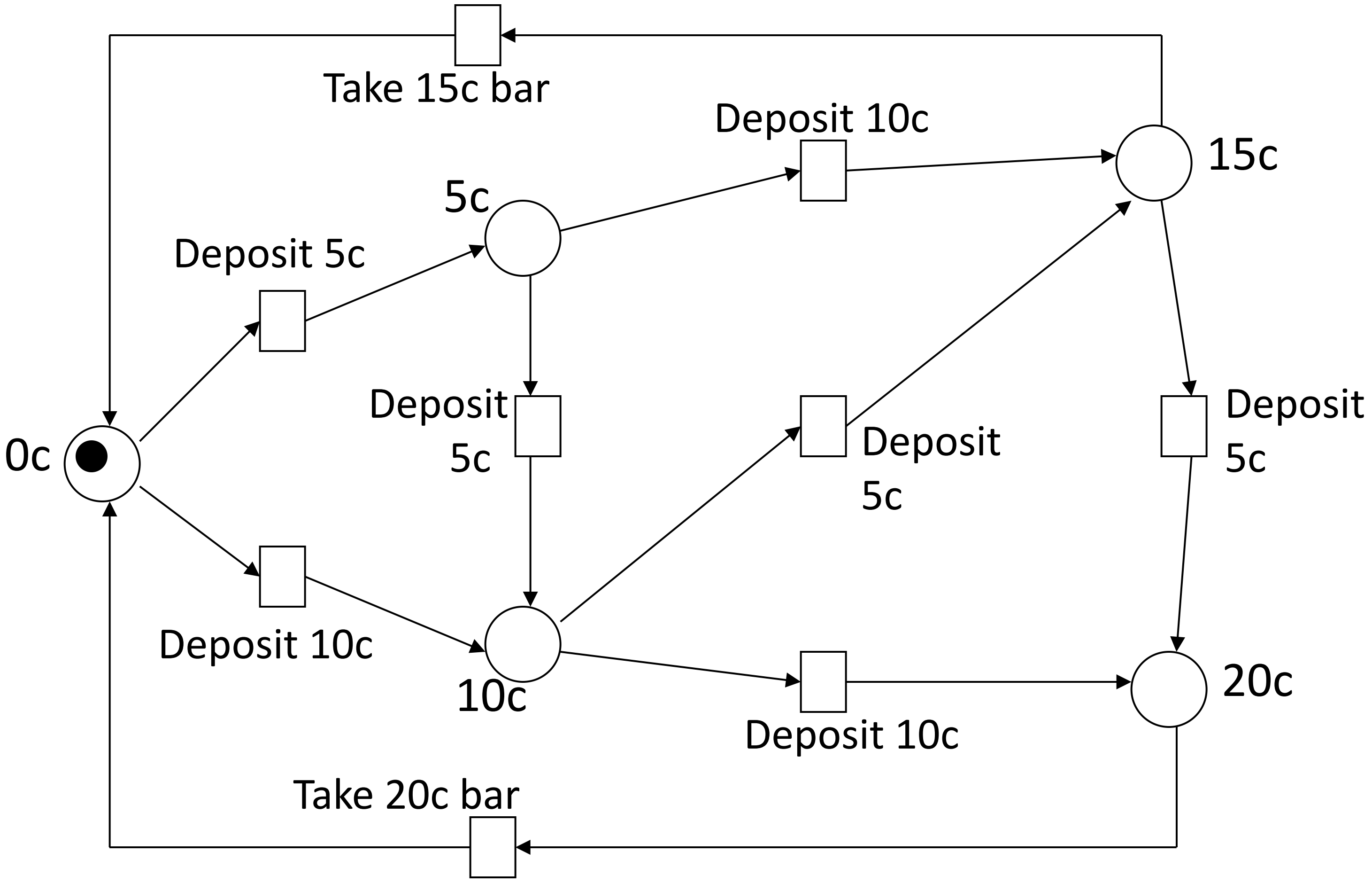
A Change of State …

- is denoted by a movement of *token(s)* (black dots) from place(s) to place(s); and is caused by the *firing* of a transition.
- The firing represents an occurrence of the event or an action taken.
- The firing is subject to the input conditions, denoted by token availability.
- A transition is *firable* or *enabled* when there are sufficient tokens in its input places.
- After firing, tokens will be transferred from the input places (old state) to the output places, denoting the new state.

Example: Vending Machine

- The machine dispenses two kinds of snack bars – 20c and 15c.
- Only two types of coins can be used
  – 10c coins and 5c coins.
- The machine does not return any change.

Petrinet on Vending Machine



Types of petrinet

- Basic Petri Nets: Traditional Petri Nets without additional features.
- Colored Petri Nets: Petri Nets with the ability to model complex data structures.
- Timed Petri Nets: Petri Nets with temporal constraints on transitions.
- Stochastic Petri Nets: Petri Nets with probabilistic transitions.
- Hybrid Petri Nets: Petri Nets combining features of different types, such as time and stochasticity.

Petri Net Modeling Process

- provides a structured approach for creating Petri Net models to represent the behavior of concurrent systems accurately.
- Problem Definition: Clearly define the system to be modeled and its objectives.
- Identification of Components: Identify the places, transitions, and arcs needed to model the system's behavior.
- Construction of Petri Net Model: Create the Petri Net diagram based on the identified components and their interactions.
- Validation and Analysis: Validate the Petri Net model for correctness and analyze its behavior using simulation or formal verification techniques.
- Iterative Refinement: Refine the model based on analysis results and feedback, if necessary, to improve its accuracy and effectiveness.

Petri Net Analysis Techniques

- to evaluate system behavior and properties. These techniques help uncover potential issues, verify system correctness, and optimize system performance.
- Reachability Analysis: Determines whether a certain state of the system can be reached from the initial state.
- Deadlock Analysis: Identifies states in which the system cannot progress due to resource contention or conflicts.
- Liveness Analysis: Determines whether the system can eventually reach a state where all processes can proceed.
- Performance Evaluation: Assesses the performance characteristics of the system, such as throughput, latency, and resource utilization.
- Structural Analysis: Analyzes the structure of the Petri Net model to identify patterns or properties.

- Consider a Petri Net representing a simple workflow system with two places (P1 and P2) and two transitions (T1 and T2). Initially, P1 contains three tokens, and P2 is empty. When T1 fires, it removes two tokens from P1 and adds one token to P2. When T2 fires, it removes one token from P2 and adds two tokens to P1. Determine the state of the system after executing the following sequence of transitions: T1, T2, T1.

- Initially, P1 = 3, P2 = 0
- After T1 fires: P1 = 1, P2 = 1
- After T2 fires: P1 = 3, P2 = 0
- After T1 fires again: P1 = 1, P2 = 1
- Therefore, the final state of the system is: P1 = 1, P2 = 1.

Advantages of Petri Nets

- Modeling Simplicity: Petri Nets provide a graphical and intuitive way to model systems, making complex systems easier to understand and analyze.
- Formal Analysis: Petri Nets offer formal methods for system analysis, including reachability analysis, deadlock detection, and performance evaluation, ensuring the correctness and efficiency of system designs.
- Concurrency Representation: Petri Nets effectively represent concurrency and synchronization in systems, allowing for the modeling of parallelism and distributed systems.
- Visualization: Petri Nets offer visual representations that aid in communication and collaboration among stakeholders, facilitating design discussions and decision-making processes.
- Tool Support: Various software tools and libraries are available for Petri Net modeling, simulation, and analysis, enhancing productivity and enabling automation in system design and verification processes.

Disadvantages of Petri Nets

- Complexity for Large Systems: Petri Nets may become complex and difficult to manage for large-scale systems, leading to challenges in model creation, analysis, and maintenance.

- State Explosion Problem: In certain cases, Petri Nets may suffer from the state explosion problem, where the number of states grows exponentially, making analysis computationally expensive and impractical.

- Lack of Temporal Representation: Petri Nets primarily focus on modeling system states and transitions, lacking explicit support for representing temporal aspects such as timing constraints and real-time behavior.

Applications

- Manufacturing Systems: Petri Nets are widely used in modeling and analyzing manufacturing systems, including production lines, workflow processes, and resource allocation, to improve efficiency and optimize resource utilization.
- Communication Protocols: Petri Nets are employed in modeling and verifying communication protocols, such as networking protocols and distributed systems, to ensure reliable and efficient data transmission and message routing.
- Software Engineering: Petri Nets find applications in software engineering for modeling software behavior, software architecture, and workflow processes, aiding in software design, validation, and debugging.
- Biological Systems: Petri Nets are utilized in modeling biological systems, such as biochemical pathways, gene regulatory networks, and cell signaling processes, to understand biological phenomena and analyze system dynamics.
- Concurrent and Parallel Systems: Petri Nets are extensively used in modeling concurrent and parallel systems, including concurrent programming, multitasking operating systems, and parallel computing architectures, to study system behavior and performance.