SNS COLLEGE OF ENGINEERING

Kurumbapalayam (po), Coimbatore – 641 107

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

23ITT203- OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT 2

Formal System Specification in Object-Oriented Software Engineering (OOSE)

Formal System Specification refers to the use of mathematical models and formal languages to describe the system's behavior, functionality, and structure in a precise, unambiguous way. The goal of formal specification is to ensure that the system's design meets the requirements without any contradictions or ambiguities, thus improving the overall quality and correctness of the system. Formal specifications are typically used in safety-critical systems, where errors could lead to catastrophic outcomes (e.g., medical systems, aerospace applications, and banking systems).

In **Object-Oriented Software Engineering (OOSE)**, formal methods help specify the behavior and interactions of objects in the system by using mathematical representations.

1. What is Formal System Specification?

A **Formal System Specification** is a method of representing the system's design using formal mathematical models. These models precisely define the system's components, the relationships between them, and the expected behavior of the system under various conditions.

- **Formal**: Refers to the use of rigorous mathematical logic.
- **Specification**: Refers to the description of system functionality, structure, and behavior.

2. Benefits of Formal System Specification

1. **Unambiguous**: Formal specifications remove ambiguity and vagueness, ensuring that every requirement is clearly defined.
2. **Precise**: Mathematical models provide a precise description of system behavior, reducing misunderstandings.
3. **Verification and Validation**: Formal methods enable the verification and validation of system properties, ensuring that the design fulfills the required functionality and adheres to constraints.
4. **Detecting Errors Early**: By using formal models early in the development cycle, errors can be detected and rectified before actual implementation begins.
5. **Consistency**: Formal specification helps ensure consistency across various components of the system, reducing the risk of contradictions.

3. Formal Methods in Object-Oriented Software Engineering

In Object-Oriented Software Engineering, formal system specification can be used in various stages of the software development lifecycle, including design, analysis, and verification.

3.1 Types of Formal Methods

There are various **formal methods** that can be applied to system specification:

1. **Finite State Machines (FSM)**:
    - A **Finite State Machine (FSM)** is a mathematical model of computation that describes the system's states and the transitions between those states.
    - Used to model the behavior of systems that can be in one of a finite number of states, and where each state has associated transitions triggered by inputs.

    Example:
    In a **Library Management System**, the FSM might describe the states a book can be in:

    - **States**: Available, Borrowed, Reserved
    - **Transitions**: Borrow, Return, Reserve
2. **Petri Nets**:
    - **Petri Nets** are graphical models used to represent concurrent systems. They are used to model workflows, system processes, and dependencies.
    - Petri Nets represent the system as a set of places, transitions, and tokens that move between places according to certain rules.

    Example:
    In a **Library Management System**, a Petri Net could model the process of borrowing a book, where places represent available books, borrowed books, and user accounts, while transitions represent borrowing actions and updates to the system.

3. **Z-Notation**:
    - **Z-Notation** is a formal specification language used to describe systems using set theory and first-order predicate logic. It is often used to describe the state and operations of a system mathematically.
    - Z-Notation uses a combination of schema and predicates to specify system components and their relationships.

    Example:
    A **Library Management System** could be described using Z-Notation to model the data schema for **Books** and **Users**, and formalize the operations like **borrow** and **return**.

4. **B-Method**:
    - The **B-Method** is a formal method used to describe software systems using abstract machines. It provides a way to model the system's data and operations with a high level of precision.

    - The B-Method is particularly useful in ensuring the correctness of systems with complex behaviors or strict requirements.

  Example:
  In a **Library Management System**, the B-Method could specify the data structures (e.g., book catalog, user account) and operations (e.g., borrow, return) using formal specifications.

4. Key Concepts in Formal System Specification

4.1 State-based Specification

State-based formal methods focus on describing the system in terms of **states** and **state transitions**. The system is represented by a finite set of states, and the transitions between these states are triggered by events or conditions.

- **State**: A specific condition or situation of the system at a given time.
- **Transition**: The movement from one state to another, triggered by an event.

Example:
In a **Library Management System**, a state-based specification might describe the following:

- **States**:
    - **Idle**: The system is waiting for a user to take an action.
    - **Searching**: The system is searching for a book based on user input.
    - **Borrowing**: A user is borrowing a book from the library.
- **Transitions**:
    - From **Idle** to **Searching** when the user initiates a book search.
    - From **Searching** to **Idle** when the search is complete.
    - From **Idle** to **Borrowing** when a user chooses to borrow a book.

4.2 Event-based Specification

Event-based specification describes how the system responds to **events** or **inputs**. The focus is on identifying events that trigger changes in the system and how those changes are handled.

- **Event**: A significant occurrence that can trigger a change in the system.
- **Response**: The system's reaction to an event.

Example:
In a **Library Management System**, an event-based specification might describe:

- **Event**: User requests to borrow a book.
- **Response**: Check if the book is available, then update the status of the book to "borrowed" and deduct the borrowing limit of the user.

4.3 Process-based Specification

Process-based specification is used to model **sequences of events** or **activities** that occur in the system. This method helps describe workflows, interactions, and dependencies between processes.

Example:
A **Library Management System**'s process-based specification could describe the process for borrowing a book:

1. **Search**: User searches for a book.
2. **Availability Check**: The system checks if the book is available.
3. **Borrowing Process**: If available, the system allows the user to borrow the book, updates the book's status, and records the transaction.

5. Examples of Formal System Specification

Example 1: Finite State Machine (FSM)

Consider a **Library Management System** where a book can be in the following states:

- **Available**
- **Borrowed**
- **Reserved**

States:

- **Available**: Book is not borrowed.
- **Borrowed**: Book is checked out by a user.
- **Reserved**: Book is reserved by a user but not yet borrowed.

Transitions:

- **Borrow**: From **Available** to **Borrowed** when a user borrows the book.
- **Return**: From **Borrowed** to **Available** when the user returns the book.
- **Reserve**: From **Available** to **Reserved** when the user reserves the book.

Example 2: Petri Net

A **Petri Net** for a **Library Management System** might represent the process of borrowing a book:

- **Places**: Represent the current status of books (e.g., available, borrowed).
- **Transitions**: Represent actions (e.g., borrow, return).
- **Tokens**: Represent the instances of books and their availability.

Formal system specification is an essential practice in systems engineering, especially when designing complex or critical systems. It ensures the system behaves as expected and helps in

verifying the correctness of the system before implementation. By using mathematical models such as **Finite State Machines**, **Petri Nets**, **Z-Notation**, and **B-Method**, engineers can precisely define system behavior, identify potential errors, and ensure system consistency.

In **Object-Oriented Software Engineering**, formal methods can be integrated into the analysis and design phases, aligning well with object-oriented principles like **classes**, **objects**, and **interactions**. Using formal specification methods in OOSE can lead to better-defined, more reliable software systems.

Finite State Machines (FSM) in Object-Oriented Software Engineering (OOSE)

A **Finite State Machine (FSM)** is a conceptual model used to represent the behavior of a system in terms of states, events, and transitions between states. It is particularly useful in designing systems that exhibit a finite number of states and whose behavior is triggered by specific events or conditions.

1. Key Concepts of FSM

- **States**: These are the various conditions or modes that the system can be in. At any given point, the system is in exactly one state.
  - **Example**: In a **Turnstile**, the states could be "Locked" and "Unlocked".
- **Transitions**: These define how the system moves from one state to another. Transitions occur when certain **events** happen.
  - **Example**: A transition from "Locked" to "Unlocked" occurs when a user inserts a coin.
- **Events**: These are external inputs that trigger a transition between states. Events are the cause of state changes.
  - **Example**: An event could be the **insertion of a coin** or a **button press**.
- **Initial State**: This is the state in which the system starts when it is first activated.
  - **Example**: A turnstile typically starts in the "Locked" state.
- **Final State** (optional): Some FSMs have final states, which represent the end of a process or activity.
  - **Example**: A **file download system** may have a final state called **"Download Completed"**.

2. Types of FSMs

1. **Deterministic Finite State Machine (DFSM)**:
   - In a DFSM, each state has exactly one transition for each possible event. This makes the behavior predictable and unambiguous.
   - **Example**: A simple **Turnstile FSM**, where inserting a coin always unlocks the turnstile.
2. **Non-Deterministic Finite State Machine (NDFSM)**:
   - In an NDFSM, a state can have multiple transitions for the same event. This introduces multiple possible outcomes, allowing for more flexible behavior.

- **Example**: In a **Game System**, a character might have multiple actions that can be triggered by the same button press, depending on the current game state (e.g., standing, running, or jumping).

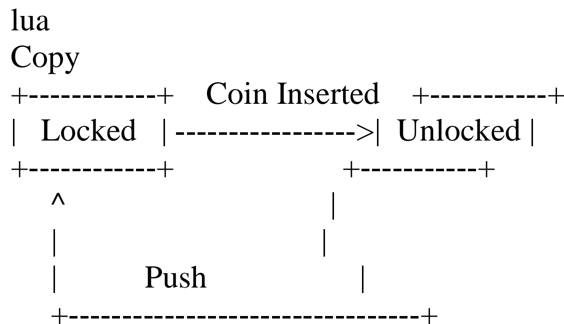
3. FSM Diagram Representation

An **FSM Diagram** visually represents the states and transitions in a system. States are typically represented as **circles**, and transitions are represented as **arrows** between the circles. Each arrow is labeled with the event that triggers the transition.

Example: Simple Turnstile FSM

- **States**:
  - **Locked**: The turnstile is locked and prevents entry.
  - **Unlocked**: The turnstile is unlocked and allows entry.
- **Events**:
  - **Coin Inserted**: A coin is inserted by the user.
  - **Push**: The user attempts to push the turnstile.
- **Transitions**:
  - From **Locked** to **Unlocked** when a **coin is inserted**.
  - From **Unlocked** to **Locked** when the user **pushes** the turnstile.

FSM Diagram:



lua
Copy
```
+------------+    Coin Inserted    +-----------+
|  Locked   | ----------------->|  Unlocked |
+------------+                   +-----------+
    ^                           |
    |                           |
    |         Push              |
    +--------------------------------+
```

4. FSM in Object-Oriented Systems

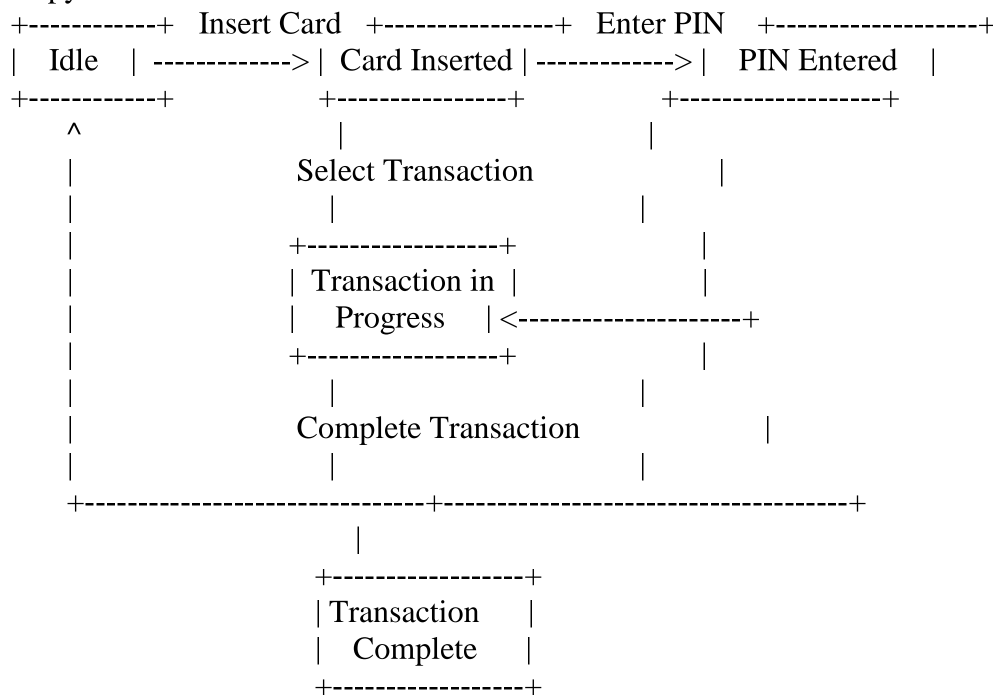
In **Object-Oriented Software Engineering (OOSE)**, FSMs can be used to model the lifecycle and behavior of objects in different states. FSMs can help describe how objects should behave depending on the state they are in.

Example: ATM Machine

- **States** of the ATM machine:
  - **Idle**: The ATM is waiting for the user to insert a card.
  - **Card Inserted**: The ATM has detected the card and is awaiting the PIN.

- **PIN Entered**: The user has entered the correct PIN and is ready for transactions.
  - **Transaction in Progress**: The ATM is processing a transaction, like a withdrawal.
  - **Transaction Complete**: The transaction has been completed, and the ATM returns to the **Idle** state.
- **Events**:
  - **Insert Card**: The user inserts a card into the ATM.
  - **Enter PIN**: The user enters the PIN.
  - **Select Transaction**: The user selects a transaction (e.g., withdrawal).
  - **Complete Transaction**: The transaction is finished.
- **FSM Diagram**:

pgsql
Copy



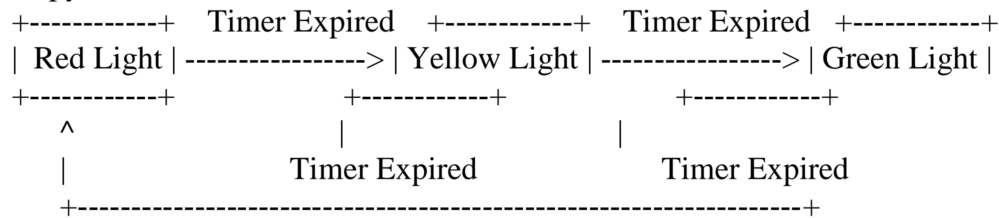
5. Real-Life Example of FSMs

Example 1: Traffic Light System

A **Traffic Light System** can be modeled using FSMs with the following states:

- **Red Light**: The light is red, and vehicles must stop.
- **Yellow Light**: The light is yellow, and vehicles should prepare to stop.
- **Green Light**: The light is green, and vehicles can go.
- **Events**:
  - **Timer Expiry**: The timer expires and triggers a change in the light color.
- **FSM Diagram**:

lua
Copy

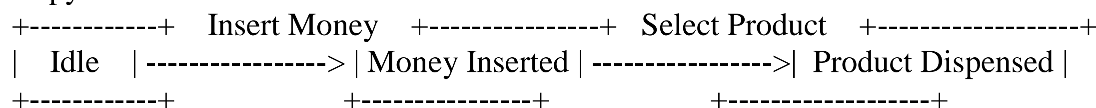


Example 2: Vending Machine

A **Vending Machine** has different states based on user interaction:

- **Idle**: The machine is waiting for the user to insert money.
- **Money Inserted**: The machine has received money but is waiting for the user to select a product.
- **Product Dispensed**: The machine dispenses the product and returns to the **Idle** state.
- **Events**:
  - **Insert Money**: The user inserts money into the machine.
  - **Select Product**: The user selects a product.
  - **Dispense Product**: The machine dispenses the product and returns to **Idle**.
- **FSM Diagram**:

mathematica
Copy



6. Why Use FSMs in Object-Oriented Design?

FSMs are useful in object-oriented design for the following reasons:

1. **State Management**: FSMs help manage the **states** of an object or system and define how the system should behave in each state.
2. **Clear Structure**: FSMs provide a **clear, structured** way to visualize and define the lifecycle and behavior of objects.
3. **Event-Driven**: FSMs align well with **event-driven programming**, where certain actions trigger specific behaviors in a system.
4. **Predictability**: FSMs help ensure that system behavior is **predictable** and consistent, especially in complex scenarios.
5. **Flexibility**: FSMs are flexible, allowing for the easy addition of new states and transitions as the system evolves.

7. Conclusion

Finite State Machines (FSM) are a powerful concept in Object-Oriented Software Engineering for modeling systems where the behavior is defined by a limited set of states and transitions. FSMs are widely used to design systems that respond to events and exhibit different behaviors depending on their current state. They are especially useful for creating systems with clear, predictable behavior, such as **user interfaces**, **game mechanics**, and **workflow processes**.

FSMs provide a structured, formal way to model the system's behavior, making it easier to understand, design, and implement complex systems.

Petri Nets in Object-Oriented Software Engineering (OOSE)

Petri Nets are a mathematical modeling tool used to describe and analyze the behavior of systems that exhibit concurrent, asynchronous, distributed, or parallel behavior. In Object-Oriented Software Engineering (OOSE), Petri Nets are often used to model the flow of control and data, particularly in systems that involve multiple processes or interactions.

1. What are Petri Nets?

A **Petri Net** is a graphical and mathematical tool that represents a system as a network of places, transitions, and arcs. It is particularly useful for modeling systems that have:

- **Concurrency**: Multiple processes or tasks occurring simultaneously.
- **Synchronization**: Coordination between processes or tasks.
- **Communication**: Data transfer between different parts of the system.

Petri Nets are widely used in various domains, including manufacturing systems, communication protocols, and software systems.

2. Key Elements of Petri Nets

A Petri Net consists of the following basic components:

1. **Places**:
    - Represent the states or conditions in the system.
    - Typically depicted as **circles**.
    - **Example**: In a **Document Workflow System**, places could represent different stages, such as "Document Drafting", "Document Review", and "Document Approved".
2. **Transitions**:
    - Represent the events or actions that cause changes in the system.
    - Typically depicted as **rectangles** or **bars**.
    - **Example**: A transition could represent an action like "Submit for Review" or "Approve Document".
3. **Tokens**:

- o Represent the current state or condition of the system and are placed in the places.
- o The presence or absence of a token in a place indicates whether the system is in a particular state.
- o **Example**: A token in the "Document Drafting" place indicates that the document is being drafted.

4. **Arcs**:
    - Represent the relationships between places and transitions.
    - Arcs connect **places** to **transitions** and **transitions** to **places**, showing how the system evolves.
    - **Example**: An arc from the "Document Drafting" place to the "Submit for Review" transition would represent that the drafting process can trigger the submission for review.

3. Petri Net Diagram

A **Petri Net diagram** is a graphical representation of a system's components and their interactions. In a Petri Net diagram:

- **Circles (places)** represent conditions or states.
- **Rectangles or bars (transitions)** represent events or actions that trigger state changes.
- **Arcs** represent the flow of tokens between places and transitions.

Example: Simple Petri Net for Document Workflow

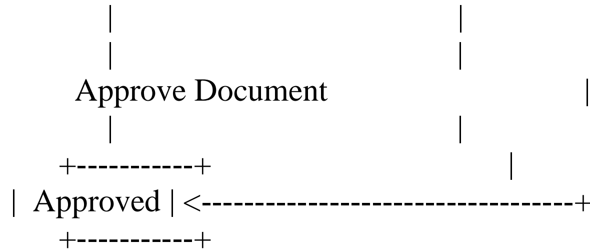
Let's consider a simple **Document Workflow System** where a document goes through various stages:

1. **Document Drafting**
2. **Document Review**
3. **Document Approval**

- **Places**:
    - "Drafting"
    - "Under Review"
    - "Approved"
- **Transitions**:
    - "Submit for Review"
    - "Approve Document"

Petri Net Representation:

lua
Copy



In this Petri Net:

- Tokens would initially reside in the "Drafting" place when the document is being created.
- The transition **Submit for Review** moves the token from "Drafting" to "Under Review" when the document is ready to be reviewed.
- The transition **Approve Document** moves the token from "Under Review" to "Approved" once the document is approved.

4. Types of Petri Nets

1. **Place/Transition (P/T) Nets**:
   - The most basic and common form of Petri Nets.
   - In P/T nets, places hold tokens, and transitions fire when the required number of tokens are in the input places.
2. **Colored Petri Nets (CPN)**:
   - A more advanced version of Petri Nets that allows tokens to have different **colors** (types), enabling the representation of more complex systems.
   - Useful for systems that involve different types of data or resources.
3. **Timed Petri Nets**:
   - A variant where transitions are associated with time, representing systems where timing constraints are important.
   - Useful for modeling time-dependent systems, such as real-time software.

5. Characteristics of Petri Nets

- **Concurrency**: Petri Nets can model concurrent operations, where multiple transitions can occur simultaneously.
  - **Example**: In a **manufacturing process**, multiple machines can operate in parallel, and Petri Nets can model this concurrent behavior.
- **Synchronization**: Petri Nets can represent systems where transitions need to synchronize with each other.
  - **Example**: In a **multi-threaded software system**, multiple processes may need to wait for each other to complete before proceeding.
- **Conflict**: Petri Nets can model situations where resources are limited and different processes may need to compete for the same resource.
  - **Example**: In a **database system**, multiple users may compete to access the same record.

- **Deadlock**: Petri Nets can help detect deadlocks, where processes are stuck in a state, waiting for resources that never become available.
  - **Example**: In a **network protocol**, two processes could wait for each other indefinitely, causing a deadlock.

6. How Petri Nets are Used in Object-Oriented Software Engineering

In OOSE, Petri Nets can be used for several purposes:

1. **Modeling Concurrent Systems**: Petri Nets are particularly useful for modeling **concurrent processes** or multi-threaded systems, where multiple events can happen at the same time, and the system needs to manage synchronization and conflicts between processes.
   - **Example**: A **banking system** might have several concurrent processes like balance checks, withdrawal requests, and deposit processes. Petri Nets can help model these interactions.
2. **Workflow Modeling**: Petri Nets are effective for describing and analyzing workflows and processes that require synchronization and coordination between different steps.
   - **Example**: A **business process** like processing an insurance claim can be modeled using a Petri Net, with places representing steps like "Claim Submitted", "Claim Under Review", and "Claim Approved".
3. **Communication Protocols**: Petri Nets can be used to represent communication protocols, where messages are exchanged between systems, ensuring that the communication flow is correct and synchronized.
   - **Example**: A **client-server** protocol for a file transfer system can be modeled using Petri Nets to ensure that the server responds properly to client requests.
4. **Event-Driven Systems**: Petri Nets can model systems where state changes are driven by events, making it useful for **event-driven programming**.
   - **Example**: A **software GUI** with buttons and event handlers can be modeled using Petri Nets, where button presses trigger transitions between states in the application.

7. Advantages of Using Petri Nets

- **Visual Representation**: Petri Nets provide a **clear graphical representation** of complex systems, making it easier to understand and analyze behavior.
- **Concurrency Management**: Petri Nets excel in modeling **concurrent activities** that need to be coordinated or synchronized.
- **Formal Analysis**: Petri Nets provide a **formal method** for analyzing system behavior, including detecting problems such as deadlocks or resource conflicts.
- **Versatility**: Petri Nets can be extended to handle various complexities, such as **timing** (Timed Petri Nets) and **different data types** (Colored Petri Nets).

8. Limitations of Petri Nets

- **Complexity**: For very large or highly complex systems, Petri Nets can become difficult to manage and analyze due to the large number of places, transitions, and arcs.

- **Scalability**: While Petri Nets are effective for small to medium-sized systems, scaling them to very large systems may require significant effort and computational resources.

9. Conclusion

Petri Nets provide a powerful and flexible way to model and analyze systems with concurrent, asynchronous, or distributed behavior. They are especially useful in **Object-Oriented Software Engineering** for modeling complex systems, workflows, communication protocols, and event-driven systems. Petri Nets help in understanding the flow of control, synchronization, and interactions between components, making them a valuable tool for designing and analyzing concurrent systems.