# Other testing methods: FSMs and Markov chains

- Capture sequence of steps for testing
- Allow complex interactions to be specified and tested
- Arbitrary level of abstraction
- Easy to specify in format suitable for automatic processing
- Enhanced with transition probabilities to Markov chain

# Other tools and methods for testing

- Unit testing, inspections, walkthroughs, checklist, input partitioning, cyclomatic complexity. They have limitations
- Need to faithfully capture interactions among different program modules
- Finite state machines
- Example of use of FSM for menu-based applications
- Enhancements of FSMs with Markov Chains
- Statistical testing.

Jeff Tian, "Software Quality Engineering", Chapter 10, Wiley Interscience

# Enhancing testing strategies

- Techniques seen so far can be enhanced.
- Not expressive enough to capture complex interactions among modules.
- Complex interactions appear when testing is done at the sub-module level.
- A method that allows complex interactions and abstraction is needed.
- FSM: Balance between expressive power and simplicity

# Extending the “single state” model

- So far, the correspondence between inputs and outputs is done in a single step: after executing a program unit.
- We need to capture “iterations” as part of the tested behaviour.
- Extend the notion of “stage” to the notion of “state”
- The inputs and outputs are considered as “sequences”
- Iteration is allowed.

**Use of FSM offer a more expressive testing mechanism that is more appropriate for certain applications.**

# FSMs: Basic concepts

- Used in numerous disciplines
- Elements: {States, Transitions, Inputs, Outputs}
  - States: Set S (finite)
  - Input values: Set I (finite)
  - Transitions: Relation: $S \; x \; I \; \rightarrow S$
  - Output values: Set O (finite)
    - Moore: $S \; \rightarrow O$
    - Mealy: $S \; \times I \rightarrow O$
- System is always in "current state"

# FSM to capture sequential behaviour

- State in FSM = Execution stage in the program or instance between certain actions.
- Example:
  - Program starts in the "initial" state.
  - User **clicks** in "File" menu.
  - **The File menu appears.**
  - User **clicks** in "Save" item
  - **The File menu disappears.**
  - File is saved
  - Program goes back to the "initial" state

# States are abstractions

- States capture the execution of an arbitrarily complex set of instructions.
- The amount of computation captured by a state is arbitrary
- Examples:
  - Browse through an application in the web
    - Each page visited can be considered a state
    - When a link is followed, a transition takes place
    - The outputs can be assigned to the states by checking the resulting page
  - Traverse menus in an application
    - The events are produced by the user
    - The overall aspect of the application is represented by the states.

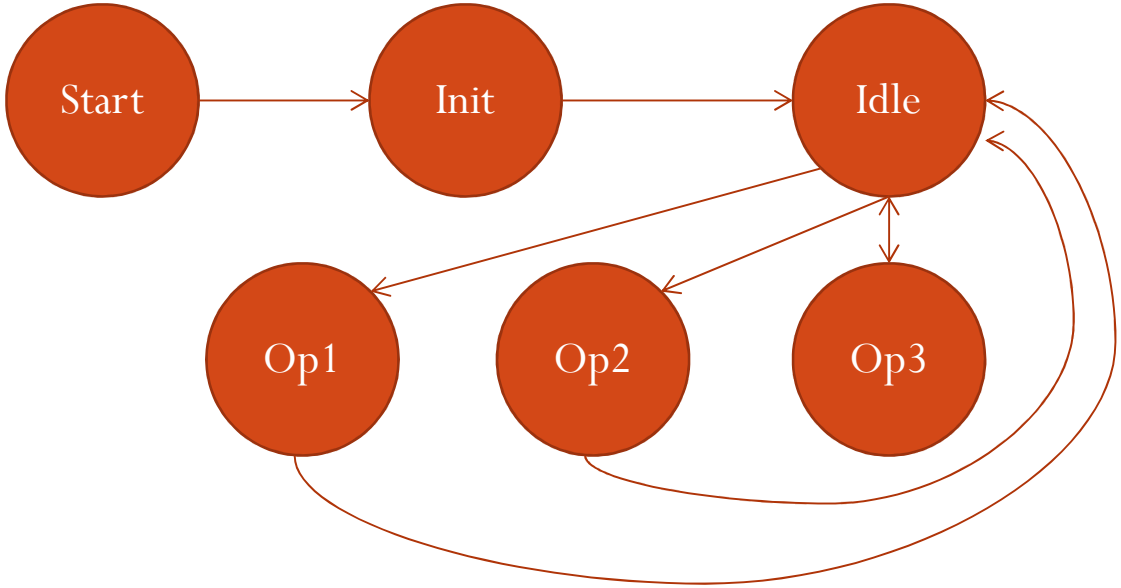
# Difference between CFG and FSMs

- CFGs
  - Control flow graphs are created within a single function.
  - Transitions represent sequences
  - No inputs are considered
  - Outputs are ignored.
  - Focus on paths.

- FSMs
  - Arbitrary functionality in one single state
  - Transitions depending on inputs
  - Outputs are taken into account
  - Focus on sequences of inputs, states and outpus.

# Additional aspects of FSMs

- States may model different levels of granularity.
- Transitions may not be associated to any input. They simply follow the normal sequence of operations.
- Transitions are **deterministic:** Given a state and an input, a single next state is possible.



# FSM Representations

- Description of the information represented by the states and the transitions.
- Small number of states
  - Graphical representation
  - Nodes and arrows
  - Easy to create
- Large number of states
  - Tabular representation of the transition and output relations
  - Set of inputs and states are implicit
  - Easy to process automatically by tools
- Large number of states, few transitions
  - Transition lists

# Example of FSM Tabular Encoding

| | Init | Op1 | Op2 | Op3 |
|---|---|---|---|---|
| **Init** | | Menu 1 | Menu 2 | |
| **Op1** | Finish | | | Menu 3 |
| **Op2** | Finish | | | |
| **Op3** | Finish | | | |

# Transition lists

- Used for highly sparse transitions
- Only a small subset of possible transitions are possible
- Typical representation: {S: ($input_1$, $output_1$, $state_1$), …($input_k$, $output_k$, $state_k$)}
- More compact
- Ideal for automatic processing

# Problems when using FSMs

- Flexible, yet with limitations.
- Rely on a robust and reliable encoding of operations in states
- Subject to changes when designs change
- Problems may derive from the four components of the machine: **states, transitions**, inputs, outputs.

# Problems with FSM states

- Incorrect states present in the FSM.
  - Incorrect state. It specifies incorrect behaviour.
  - Missing states. Certain parts of the application do not correspond with any of the states in the FSM. Example: Lack of initial state.
  - Extra states. Those states that are not reachable in the FSM or they have no transition possible to them due to some combination of inputs that are not possible in the application.

# Problems with FSM transitions

- Incorrect transitions: Connects one state to an unexpected one with unexpected output.
- Missing transitions: There are sequences of states that are note connected when the application makes a transition among them.
- Extra transitions: There are transitions in the FSM that do not correspond with anything possible in the product.

# Steps to create a FSM

- FSMs can be constructed in three steps:
  1. Information source identification and data collection
  2. Construction of the initial FSM
     1. State identification and enumeration
     2. Transition identification with the help of input values
     3. Identifying input-output relations
  3. Model refinement and validation

## Step 1. Information source identification and data collection

- Identify the sources of inputs and the required outputs.
  - Derive from functional specification, software requirements.
- Select the appropriate level of abstraction for the states
  - May need to combine various operations into single states.
  - May combine existing test cases that are lumped together into a more complex one.

# Step 2: Construction of the initial FSM

- State identification and enumeration:
  - Keep the number of states manageable. If too large, the FSM may be abstracted. If too few, more detail may be inserted by new states.
  - Some techniques such as "hierarchical FSMs" have been proposed to be able to capture flexible abstraction
- Transition identification with input values:
  - Consider all possible transitions from each state.
  - Use partitions for large sets (i.e. remaining values)
- Identify input/output relations.

# Step 3: Model refinement and validation

- Identify potential new states and/or transitions
- If too many states appear, consider hierarchical FSMs
- Validate the FSM with the functional and software requirements.
- Remove states and/or transitions that are incorrect or redundant.

# Exercising the FSM

- State coverage
  - Each state can be reached and visited by one test case.
  - State/node traversal problem.
  - Graph traversal algorithms can be used to derive the test cases.
- Transition coverage
  - Each link is covered by some test case.
  - Much more efficient when combined with the state coverage.
- Combined problem: Create the shortest number of paths in the FSM that accomplish both state and transition coverage.

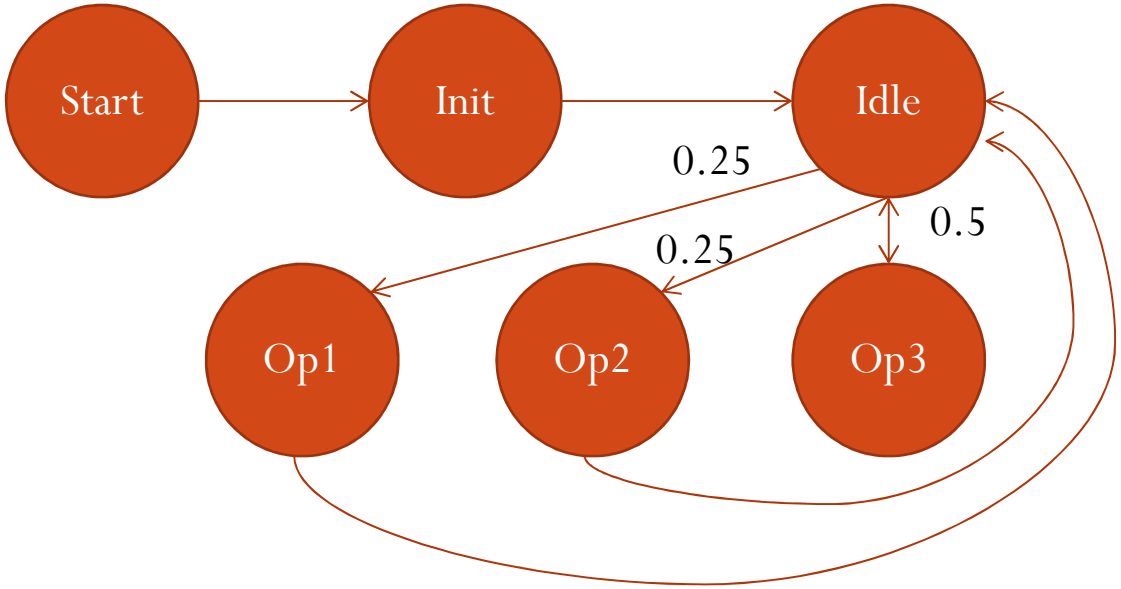
# Applications

- FSM testing is used mostly in menu-driven software and web-based applications.
- Each menu demands some input procedures and the consequence is some output that may include a new menu
- FSMs are also used for systems that operate continuously (i.e. mobile applications)
- Limitation: Number of states. Impractical for low level representation of complex systems.

# From FSMs to Markov Chains

- When the number of states is too large only a subset of all possible paths is traversed.
- Heuristic: Exercise those paths that are **more likely** to be used.
- Transitions in FSMs are extended with a **probability value.**
- $P_{i,j}$ Represents the probability of a transition from state "i" at time n to state "j" at time n+1.
- Sum of probabilities of all out-going transitions from a state are 1.
- If true for all states, FSM is a Markov Chain.

# Example



# Where do the probabilities come from?

- Initial estimations based on expert knowledge of the application.
- Estimations based on previously collected statistics
  - Include the collection of such statistics in the product (web systems)
- Combination of both.

# Power of Markov Chains for testing

- Generate state and transition coverage.
- Use a threshold to consider only values that are above the threshold.
- Ignore all the other transitions.
- Effectively devote the testing effort to the most used portion of the product.
- If the Markov chain is “stationary” the value of the stationary probability can be computed for each state and used for selecting tests.

# Testing Management Tools

- Products to support QA
- Cover all aspects of QA mixed with project management
- Potential productivity gains
- Very challenging to design

# Testing Management Tools

- Comprehensive support for SQE
- Support the entire workflow from initial steps to every day operations while product has been released.
- Handle multiple projects
- Support for:
  - Requirements
  - Test planning
  - Execution
  - Reports
  - Defect tracking
  - Project Management

# Integrated management

- All the aspects of quality assurance are tackled by a variety of tools
- Large and complex projects need integrated solutions
- Increase in productivity, time to market, quality
- Lucrative market with emerging products
- New approaches to product development and QA

# Big Players

- IBM Rational Quality Manager
- HP Quick Test Professional
- Micro Focus SilkCentral Test Manager
- Atlassian JIRA and other products
- Rally Software (Application Life cycle management: ALM)

# Requirements

- Quick definition
- Intuitive organization
- Detect duplicates
- Connect with tests
- Easy to update. Requirements that are not updated are useless.
- Strike balance between effectiveness and time devoted to it. Too little or too much translate into useless requirements.

# Software test planning

- Define the test cases
- Group them in different types
- Connect them to requirements
- Log the executions
- Assign responsibilities to specific developer/tester
- Allow collaborative creation of tests
- Easy design for manual tests
- Easy reporting of failures
- Connection with defect tracking
- Avoid duplicate tests

# Execution and results

- Schedule executions
- When manual tests, connect with project milestones
- Collect results and provide access to reporting
- Avoid duplicate executions
- Automatic detection of relevant tests to execute when a feature is modified.

# Reports

- Different information depending on the type of report
  - Daily build status
  - Pre-release status
  - Current product status
- Different types of reports depending on roles.
- Dashboards supporting developers and testers
- Estimations and predictions

# Defect Tracking

- Detailed descriptions
- Efficient triage
- Easy to report divergence (manual tests)
- Connect with project milestones
- Collaborative resolution

# Project Management

- QA is an integral part of the project
- QA issues may affect project execution
- Platform to quickly adapt milestones
- Quantify effort, progress.