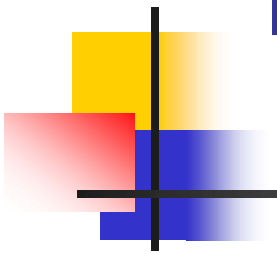
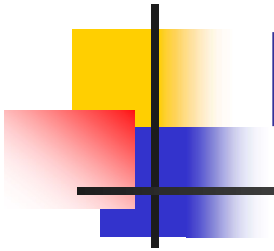
# Boundary value analysis



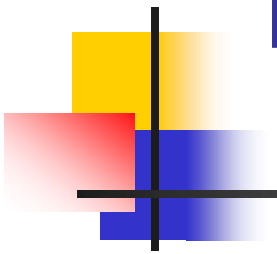
- Programs often fail on special values
- These values often lie on boundary of equivalence classes
- Test cases that have boundary values have *high yield*
- These are also called *extreme cases*
- A BV test case is a set of input data that lies on the edge of a eq class of input/output



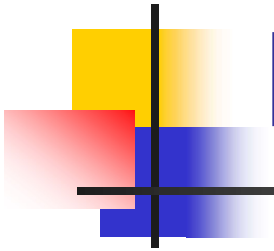
# Boundary value analysis

- Programmers may improperly use < instead of <=

# BVA...



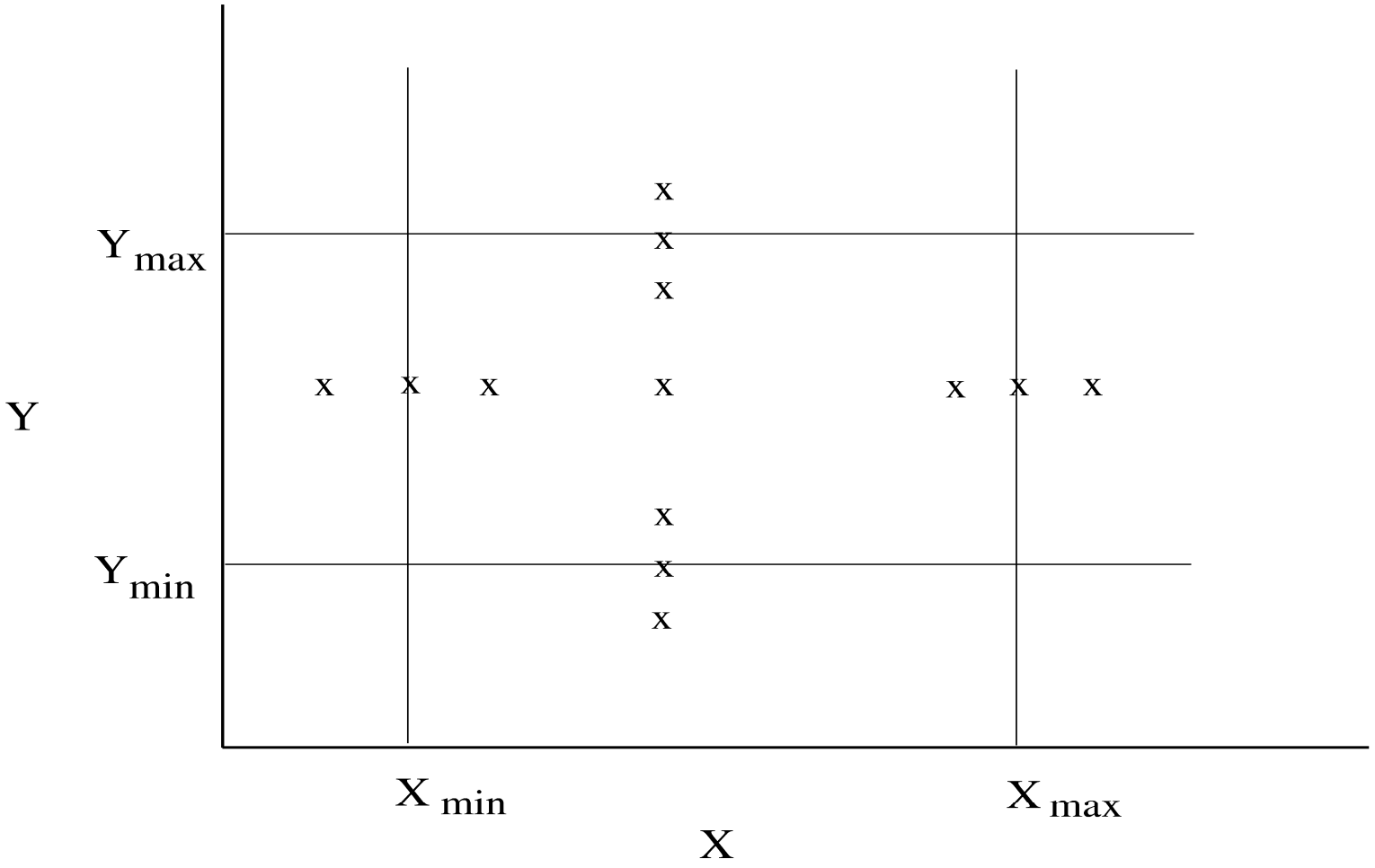
- For  each equivalence class
  - choose values on the edges of the class
  - choose values just outside the edges
- E.g. if 0 <= x <= 1.0
  - 0.0 , 1.0 are edges inside
  - -0.1,1.1 are just outside
- E.g. a bounded list - have a null list , a maximum value list
- Consider outputs also and have test cases generate outputs on the boundary

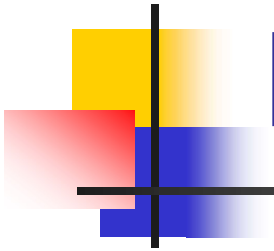


# BVA…

- In BVA we determine the value of vars that should be used
- If input is a defined range, then there are 6 boundary values plus 1 normal value (tot: 7)
- If multiple inputs, how to combine them into test cases; two strategies possible
  - Try all possible combination of BV of diff variables, with n vars this will have $7^n$ test cases!
  - Select BV for one var; have other vars at normal values + 1 of all normal values

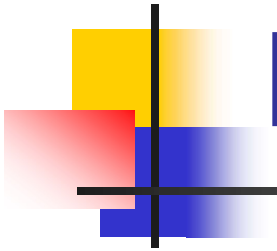
# BVA.. (test cases for two vars – x and y)





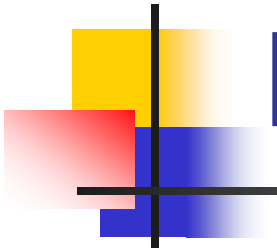
# Pair-wise testing

- Often many parmeters determine the behavior of a software system
- The parameters may be inputs or settings, and take diff values (or diff value ranges)
- Many defects involve one condition (single-mode fault), eg. sw not being able to print on some type of printer
  - Single mode faults can be detected by testing for different values of diff parms
  - If n parms and each can take m values, we can test for one diff value for each parm in each test case
  - Total test cases: m



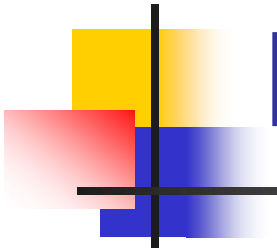
# Pair-wise testing…

- All faults are not single-mode and sw may fail at some combinations
  - Eg tel billing sw does not compute correct bill for night time calling (one parm) to a particular country (another parm)
  - Eg ticketing system fails to book a biz class ticket (a parm) for a child (a parm)
- Multi-modal faults can be revealed by testing diff combination of parm values
- This is called combinatorial testing



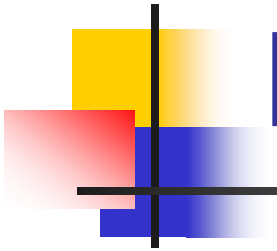
# Pair-wise testing…

- Full combinatorial testing not feasible
  - For n parms each with m values, total combinations are $n^m$
  - For 5 parms, 5 values each (tot: 3125), if one test is 5 mts, tot time > 1 month!
- Research suggests that most such faults are revealed by interaction of a pair of values
- I.e. most faults tend to be double-mode
- For double mode, we need to exercise each pair – called pair-wise testing



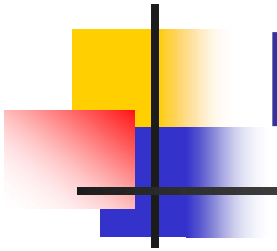
# Pair-wise testing…

- In pair-wise, all pairs of values have to be exercised in testing
- If n parms with m values each, between any 2 parms we have m*m pairs
  - 1st parm will have m*m with n-1 others
  - 2nd parm will have m*m pairs with n-2
  - 3rd parm will have m*m pairs with n-3, etc.
  - Total no of pairs are m*m*n*(n-1)/2



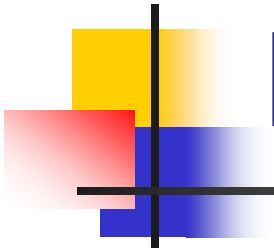
# Pair-wise testing…

- A test case consists of some setting of the n parameters
- Smallest set of test cases when each pair is covered once only
- A test case can cover a maximum of (n-1)+(n-2)+…=n(n-1)/2 pairs
- In the best case when each pair is covered exactly once, we will have $m^2$ different test cases providing the full pair-wise coverage



# Pair-wise testing…

- Generating the smallest set of test cases that will provide pair-wise coverage is non-trivial
- Efficient algos exist; efficiently generating these test cases can reduce testing effort considerably
  - In an example with 13 parms each with 3 values pair-wise coverage can be done with 15 testcases
- Pair-wise testing is a practical approach that is widely used in industry



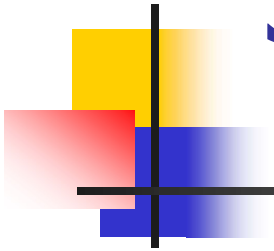
# Pair-wise testing, Example

- A sw product for multiple platforms and uses browser as the interface, and is to work with diff OSs
- We have these parms and values
  - OS (parm A): Windows, Solaris, Linux
  - Mem size (B): 128M, 256M, 512M
  - Browser (C): IE, Netscape, Mozilla
- Total no of pair wise combinations: 27
- No of cases can be less

# Pair-wise testing…

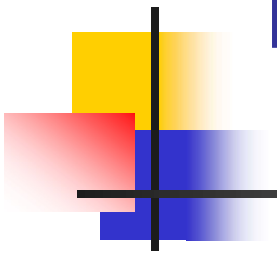
| Test case | Pairs covered |
|---|---|
| a1, b1, c1 | (a1,b1) (a1, c1) (b1,c1) |
| a1, b2, c2 | (a1,b2) (a1,c2) (b2,c2) |
| a1, b3, c3 | (a1,b3) (a1,c3) (b3,c3) |
| a2, b1, c2 | (a2,b1) (a2,c2) (b1,c2) |
| a2, b2, c3 | (a2,b2) (a2,c3) (b2,c3) |
| a2, b3, c1 | (a2,b3) (a2,c1) (b3,c1) |
| a3, b1, c3 | (a3,b1) (a3,c3) (b1,c3) |
| a3, b2, c1 | (a3,b2) (a3,c1) (b2,c1) |
| a3, b3, c2 | (a3,b3) (a3,c2) (b3,c2) |

# Special cases

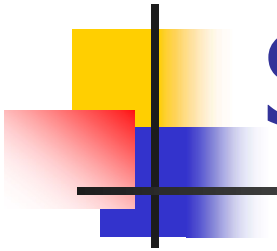


- Programs often fail on special cases
- These depend on nature of inputs, types of data structures, etc.
- No good rules to identify them
- One way  is to guess when the software might fail and create those test cases
- Also called error guessing
- Play the sadist & hit where it might hurt

# Error Guessing

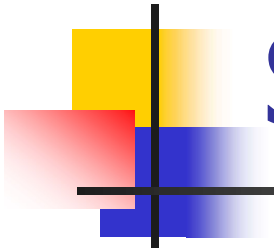


- Use experience and judgement to guess situations where a programmer might make mistakes
- Special cases can arise due to assumptions about inputs, user, operating environment, business, etc.
- E.g. A program to count frequency of words
  - file empty, file non existent, file only has blanks, contains only one word, all words are same, multiple consecutive blank lines, multiple blanks between words, blanks at the start, words in sorted order, blanks at end of file, etc.
- Perhaps the most widely used in practice



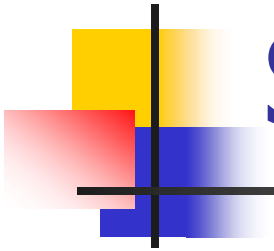
# State-based Testing

- Some systems are state-less: for same inputs, same behavior is exhibited
- Many systems' behavior depends on the state of the system i.e. for the same input the behavior could be different
- I.e. behavior and output depend on the input as well as the system state
- System state – represents the cumulative impact of all past inputs
- State-based testing is for such systems



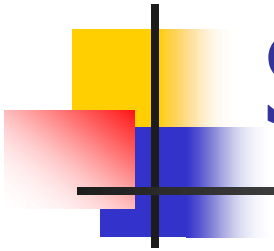
# State-based Testing…

- A system can be modeled as a state machine
- The state space may be too large (is a cross product of all domains of vars)
- The state space can be partitioned in a few states, each representing a logical state of interest of the system
- State model is generally built from such states



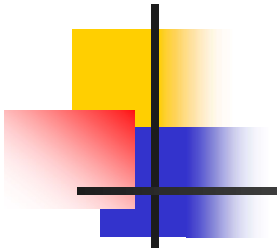
# State-based Testing…

- A state model has four components
  - States: Logical states representing cumulative impact of past inputs to system
  - Transitions: How state changes in response to some events
  - Events: Inputs to the system
  - Actions: The outputs for the events



# State-based Testing…

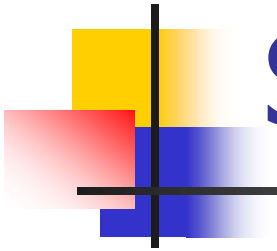
- State model shows what transitions occur and what actions are performed
- Often state model is built from the specifications or requirements
- The key challenge is to identify states from the specs/requirements which capture the key properties but is small enough for modeling



- When the tester is trying to test sequence of events that occur in the application under test. I.e., this will allow the tester to test the application behavior for a sequence of input values.
- When the system under test has a dependency on the events/values in the past.

# When to Not Rely On State Transition

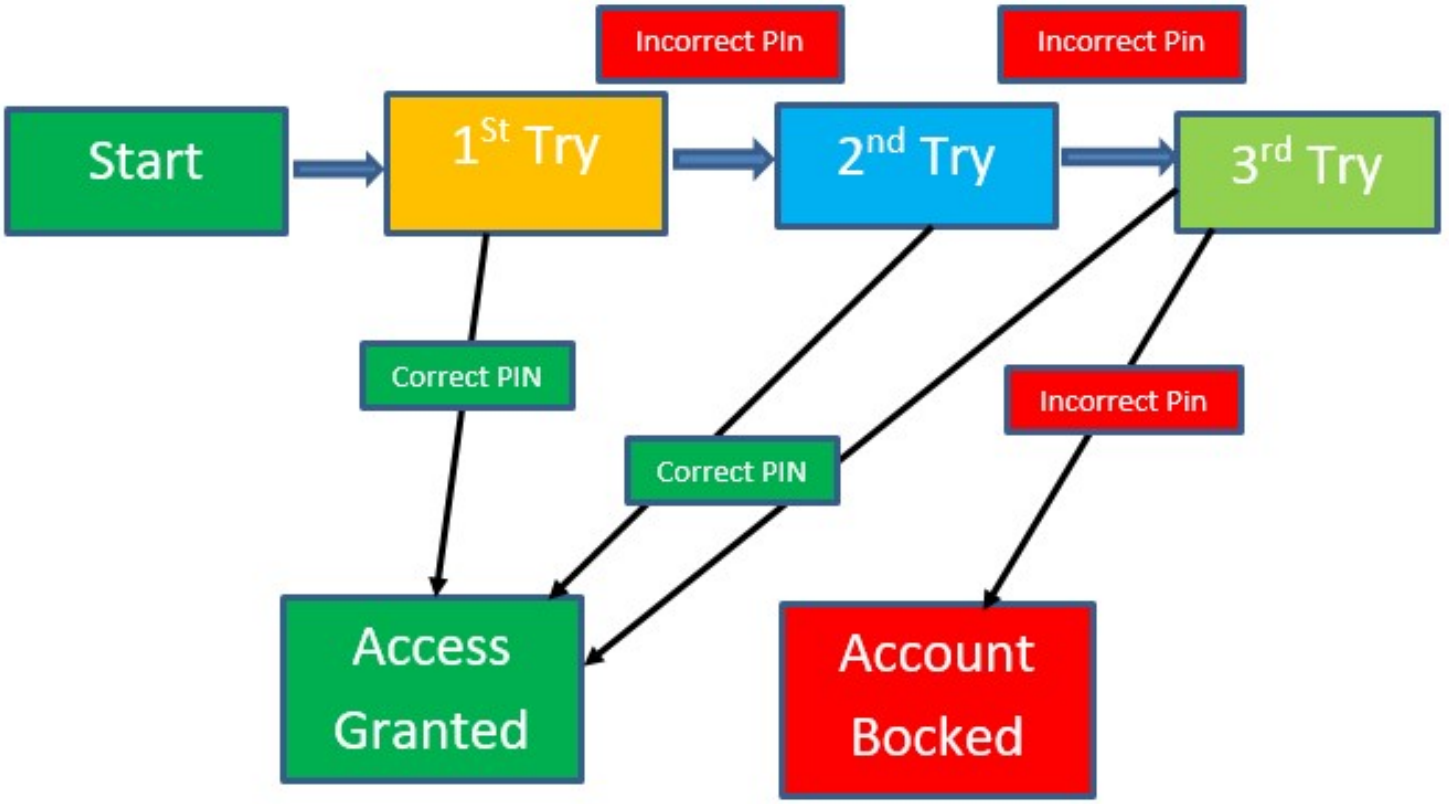
- When the testing is not done for sequential input combinations.
- If the testing is to be done for different functionalities like exploratory testing



# State-based Testing…

- State model can be created from the specs or the design
- For objects, state models are often built during the design process
- Test cases can be selected from the state model and later used to test an implementation
- Many criteria possible for test cases

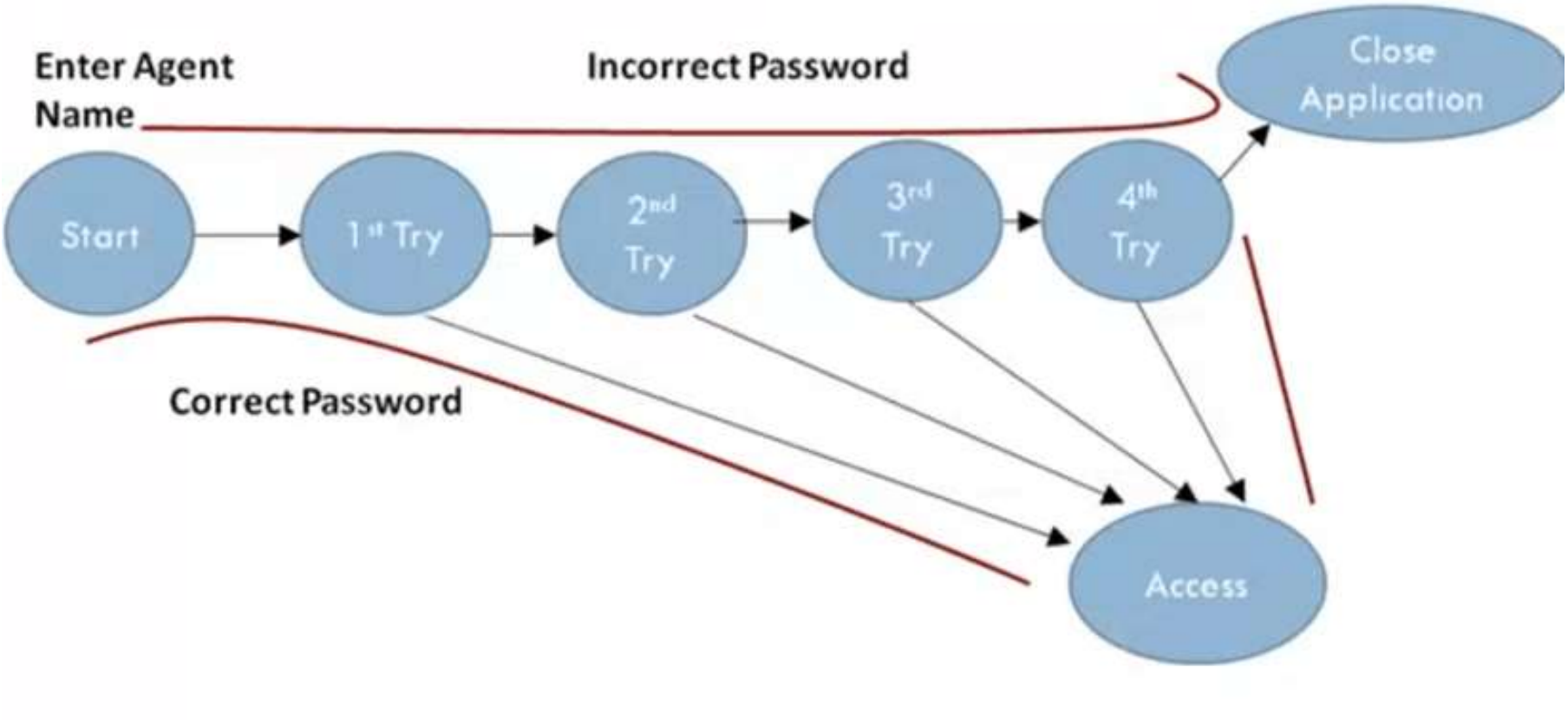
# State Transition Diagram

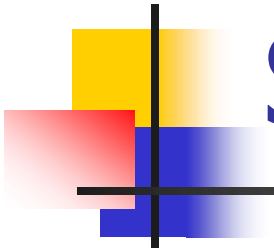


# State Transition Table

| | Correct PIN | Incorrect PIN |
|---|---|---|
| S1) Start | S5 | S2 |
| S2) 1st attempt | S5 | S3 |
| S3) 2nd attempt | S5 | S4 |
| S4) 3rd attempt | S5 | S6 |
| S5) Access Granted | - | - |
| S6) Account blocked | - | - |

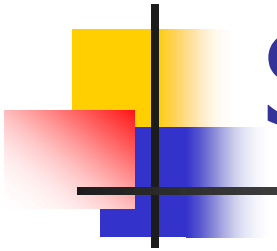
# State Transition graph





# State-based Testing criteria

- All transaction coverage (AT): test case set T must ensure that every transition is exercised
- All transitions pair coverage (ATP). T must execute all pairs of adjacent transitions (incoming and outgoing transition in a state)
- Transition tree coverage (TT). T must execute all simple paths (i.e. a path from start to end or a state it has visited)



# State-based testing…

- SB testing focuses on testing the states and transitions to/from them
- Different system scenarios get tested; some easy to overlook otherwise
- State model is often done after design information is available
- Hence it is sometimes called *grey box testing* (as it not pure black box)