

# Blackbox testing techniques

- Black-box (data-driven or input/output driven) testing is based on program specifications.
- The goal is to find **areas wherein the program does not behave** according to its specifications.
- A **good test case** is one that has a **reasonable probability of finding an error**.
- An exhaustive input test of a program is **impossible**. Hence, when testing a program, you are limited to **a small subset of all possible inputs**.
- you want to select the “right” subset, that is, the subset with the highest probability of finding the most errors.

# Equivalence Class partitioning

- Divide the input space into equivalent classes
- If the software works for a **test case from a class** then it is likely to work for all.
- Can **reduce** the set of test cases if such equivalent classes can be identified.
- Getting ideal equivalent classes is impossible
- Approximate it by identifying classes for which different behavior is specified.

# Equivalence Class Examples

*In a computer store, the computer item can have a quantity between -500 to +500. What are the equivalence classes?*

Answer: Valid class:  $-500 \leq \text{QTY} \leq +500$

Invalid class:  $\text{QTY} > +500$

Invalid class:  $\text{QTY} < -500$

# Equivalence Class Examples

*Account code can be 500 to 1000 or 0 to 499 or 2000 (the field type is integer). What are the equivalence classes?*

## **Answer:**

Valid class:  $0 \leq \text{account} \leq 499$

Valid class:  $500 \leq \text{account} \leq 1000$

Valid class:  $2000 \leq \text{account} \leq 2000$

Invalid class:  $\text{account} < 0$

Invalid class:  $1000 < \text{account} < 2000$

Invalid class:  $\text{account} > 2000$

# Equivalence class partitioning...

- Rationale: specification requires same behavior for elements in a class
- Software likely to be constructed such that it either fails for all or for none.
- E.g. if a function was not designed for negative numbers then it will fail for all the negative numbers
- For robustness, should form equivalent classes for invalid as well as valid inputs

# Equivalent class partitioning..

- Every condition specified as input is an equivalent class
- Define invalid equivalent classes also
- E.g. range  $0 < \text{value} < \text{Max}$  specified
  - one range is the valid class
  - $\text{input} < 0$  is an invalid class
  - $\text{input} > \text{max}$  is an invalid class
- Whenever that entire range may not be treated uniformly - split into classes

# Equivalence class...

- Once equivalence classes selected for each of the inputs, test cases have to be selected
  - Select each test case covering as many valid equivalence classes as possible
  - Or, have a test case that covers at most one valid class for each input
  - Plus a separate test case for each invalid class

# Example

- Consider a program that takes 2 inputs – a string **s** and an integer **n**
- Program determines **n** most frequent characters
- Tester believes that programmer may deal with diff types of chars separately
- Describe valid and invalid equivalence classes



# Example..

Input	Valid Eq Class	Invalid Eq class
S	1: Contains numbers 2: Lower case letters 3: upper case letters 4: special chars 5: str len between 0-N(max)	1: non-ascii char 2: str len > N
N	6: Int in valid range	3: Int out of range

# Example...

- Test cases (i.e.  $s$ ,  $N$ ) with first method
  - $s$ : str of  $\text{len} < N$  that includes lower case, upper case, numbers, and special chars, and  $N=5$
  - Plus test cases for each of the invalid eq classes
  - Total test cases: 1 valid+3 invalid= 4 total
- With the second approach
  - A separate string for each type of char (i.e. a str of numbers, one of lower case, ...) + invalid cases
  - Total test cases will be  $6 + 3 = 9$

# Boundary value analysis

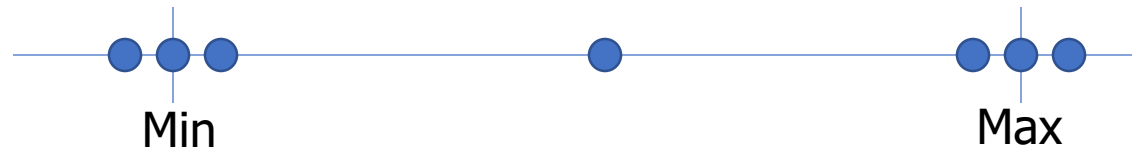
- Programs often fail on special values
- These values often lie on boundary of equivalence classes
- Test cases that have boundary values (BVs) have *high yield*
- These are also called *extreme cases*
- A BV test case is a set of input data that lies on the edge of an equivalence class of input/output

# Boundary value analysis (cont)...

- For each equivalence class
  - choose values on the edges of the class
  - choose values just outside the edges
- E.g. if  $0 \leq x \leq 1.0$ 
  - 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

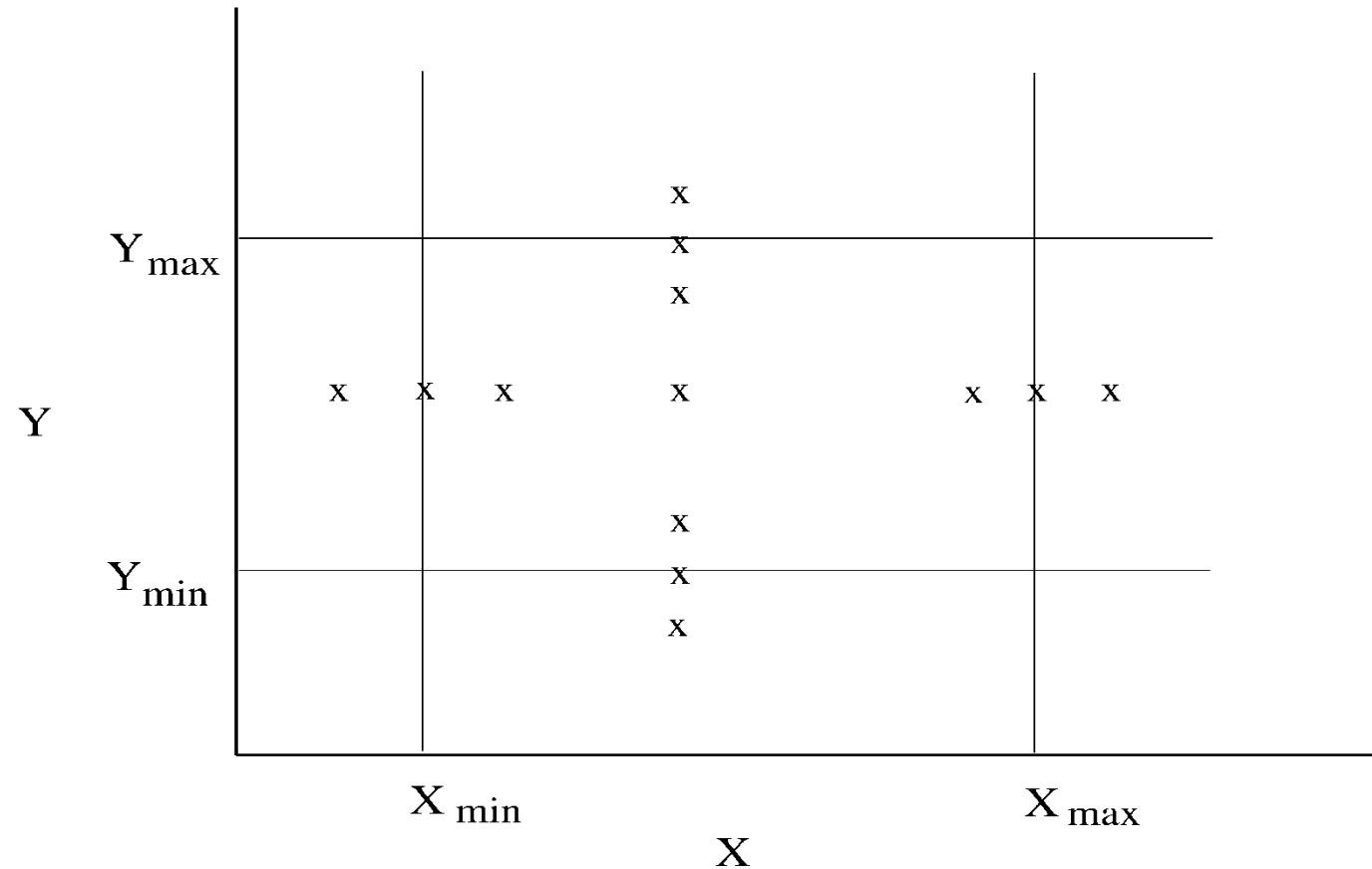
# Boundary Value Analysis

- 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)



# Cause Effect graphing

- Equivalence classes and boundary value analysis consider each input separately
- To handle multiple inputs, different combinations of equivalent classes of inputs can be tried
- Number of combinations can be large – if  $n$  diff input conditions such that each condition is valid/invalid, total:  $2^n$
- Cause effect graphing helps in selecting combinations as input conditions

# CE-graphing

- Identify causes and effects in the system
  - Cause: distinct input condition which can be true or false
  - Effect: distinct output condition (T/F)
- Identify which causes can produce which effects; can combine causes
- Causes/effects are nodes in the graph and arcs are drawn to capture dependency; and/or are allowed



# CE-graphing

- From the CE graph, can make a decision table
  - Lists combination of conditions that set different effects
  - Together they check for various effects
- Decision table can be used for forming the test cases

# Step 1: Break the specification down into workable pieces.

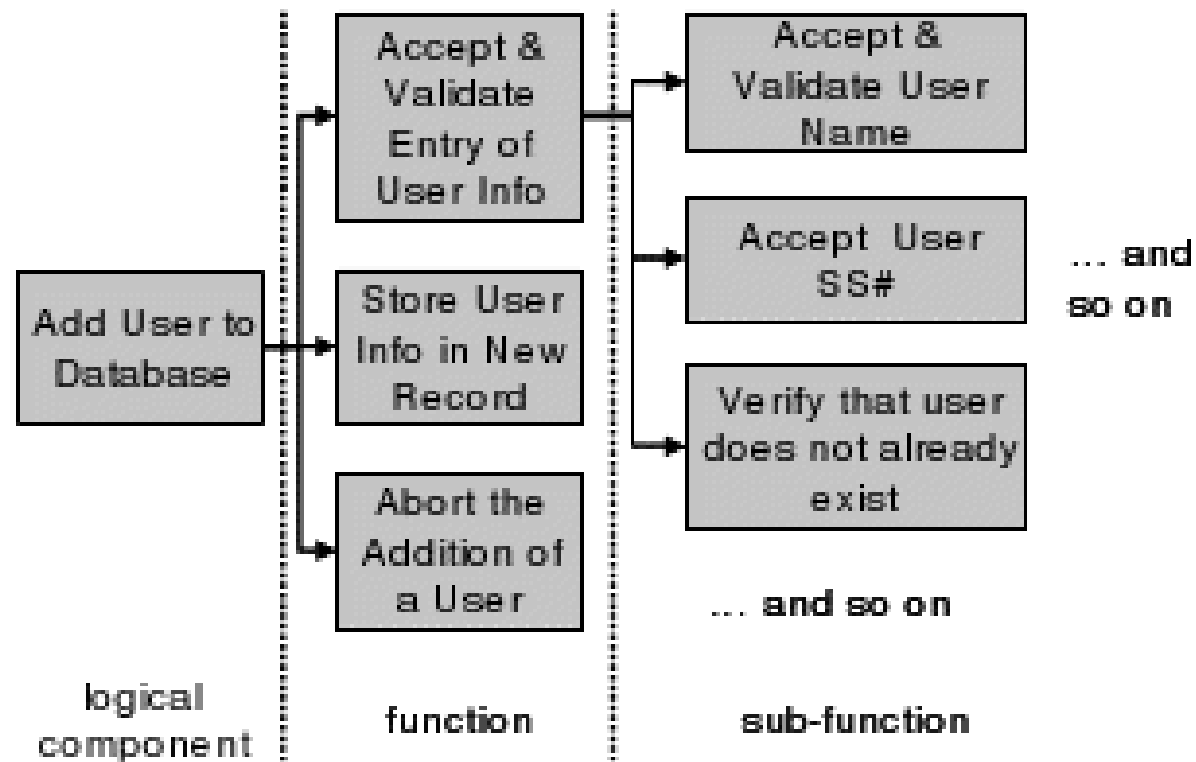


Figure 1 - Decomposed Function List

## Step 2: Identify the causes and effects.

- a) Identify the causes (the distinct or equivalence classes of input conditions) and assign each one a unique number.
- b) Identify the effects or system transformation and assign each one a unique number.

# Example

Consider the following set of requirements as an example:

Requirements for Calculating Car Insurance Premiums:

R00101 For females less than 65 years of age, the premium is \$500

R00102 For males less than 25 years of age, the premium is \$3000

R00103 For males between 25 and 64 years of age, the premium is \$1000

R00104 For anyone 65 years of age or more, the premium is \$1500

- What are the driving input variables?
- What are the driving output variables?
- Can you list the causes and the effects ?

# Example: Causes & Effects

Causes (input conditions)	Effects (output conditions)
1. Sex is Male	100. Premium is \$1000
2. Sex is Female	101. Premium is \$3000
3. Age is <25	102. Premium is \$1500
4. Age is $\geq 25$ and $< 65$	103. Premium is \$500
5. Age is $\geq 65$	

**Table 1 – Causes and Effects**

# Step 3: Construct Cause & Effect Graph

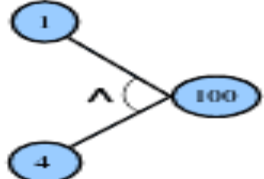
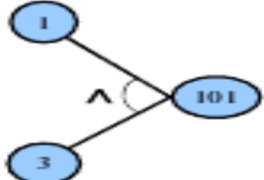
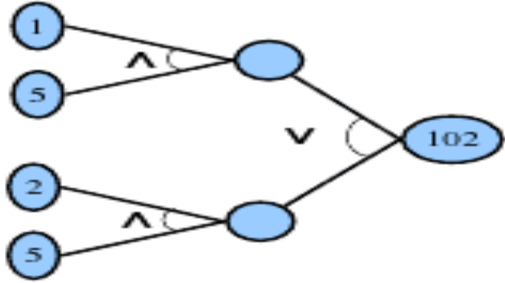
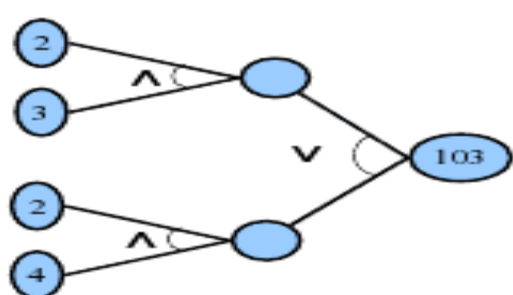
CEG	Interpretation
CEG #1: 	Causes: 1. Sex is Male and ( $\wedge$ ) 4. Age is $\geq 25$ and $< 65$ Effect: 100: Premium is \$1000
CEG #2: 	Causes: 1. Sex is Male and ( $\wedge$ ) 3. Age is $< 25$ Effect: 101: Premium is \$3000
CEG #3: 	Causes: 1. Sex is Male and ( $\wedge$ ) 5. Age is $\geq 65$ or ( $\vee$ ) 2. Sex is Female and ( $\wedge$ ) 5. Age is $\geq 65$ Effect: 102: Premium is \$1500
CEG #4: 	Causes: 2. Sex is Female and ( $\wedge$ ) 3. Age is $< 25$ or ( $\vee$ ) 2. Sex is Female and ( $\wedge$ ) 4. Age is $\geq 25$ and $< 65$ Effect: 103: Premium is \$500

Table 2 – Cause-Effect Graphs

## Step 4: Annotate the graph with constraints

- Annotate the graph with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints or considerations.
- Example: Can be both Male and Female?
- Types of constraints?
  - Exclusive: Both cannot be true
  - Inclusive: At least one must be true
  - One and only one: Exactly one must be true
  - Requires: If A implies B
  - Mask: If effect X then not effect Y

# Types of Constraints

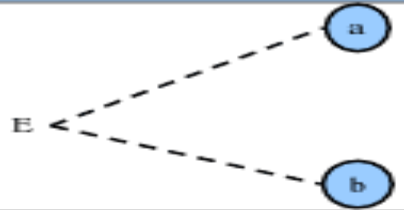
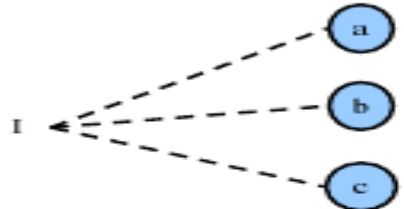
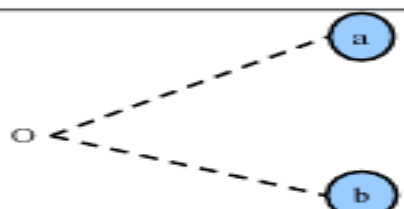

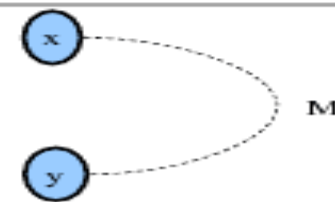
Constraint Symbol	Definition
	<p>The "E" (Exclusive) constraint states that both causes <i>a</i> and <i>b</i> cannot be true simultaneously.</p>
	<p>The "I" (Inclusive (at least one)) constraint states that at least one of the causes <i>a</i>, <i>b</i> and <i>c</i> must always be true (<i>a</i>, <i>b</i>, and <i>c</i> cannot be false simultaneously).</p>
	<p>The "O" (One and Only One) constraint states that one and only one of the causes <i>a</i> and <i>b</i> can be true.</p>
	<p>The "R" (Requires) constraint states that for cause <i>a</i> to be true, then cause <i>b</i> must be true. In other words, it is impossible for cause <i>a</i> to be true and cause <i>b</i> to be false.</p>
	<p>The "M" (mask) constraint states that if effect <i>x</i> is true; effect <i>y</i> is forced to false. (Note that the mask constraint relates to the effects and not the causes like the other constraints).</p>

Table 3 – Constraint Symbols



# Example: Adding a One-and-only-one Constraint

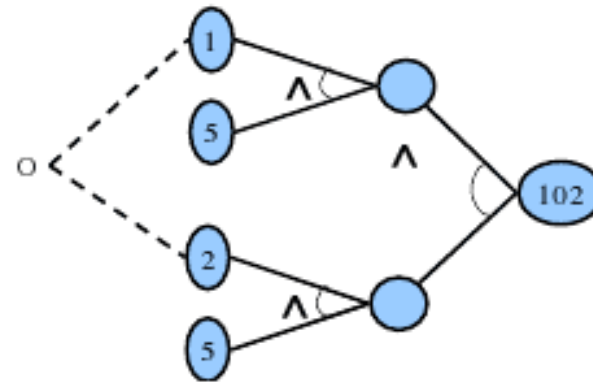
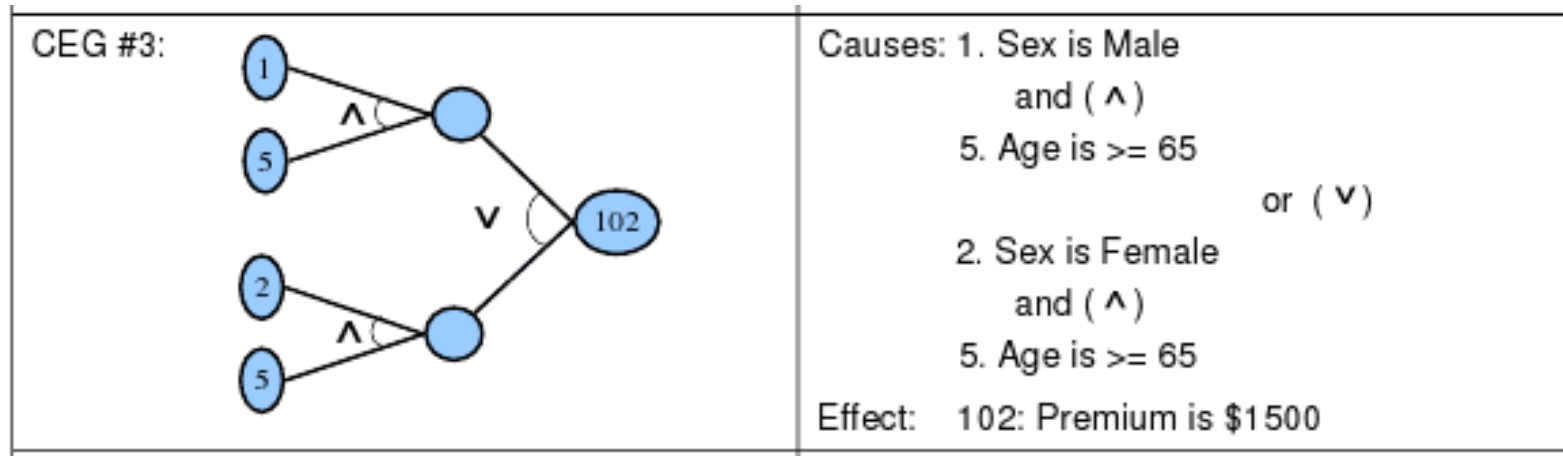


Figure 2 - Example of "O" constraint

- Why not use an exclusive constraint?

## Step 5: Construct limited entry decision table

- Methodically trace state conditions in the graphs, converting them into a limited-entry decision table.
- Each column in the table represents a test case.

Test Case	1	2	3	...	n
Cause 1	1	0	...		
...	0	1	...		
Cause c	0	0	...		
Effect 100	...	...	...		
...					
Effect e					0

# Example: Limited entry decision table

Test Case	1	2	3	4	5	6
Causes:						
1 (male)	1	1	1	0	0	0
2 (female)	0	0	0	1	1	1
3 (<25)	1	0	0	0	1	0
4 (>=25 and < 65)	0	1	0	0	0	1
5 (>= 65)	0	0	1	1	0	0
Effects:						
100 (Premium is \$1000)	0	1	0	0	0	0
101 (Premium is \$3000)	1	0	0	0	0	0
102 (Premium is \$1500)	0	0	1	1	0	0
103 (Premium is \$500)	0	0	0	0	1	1

**Table 4 – Limited-Entry Decision Table**

# Step 6: Convert into test cases

Test Case	1	2	3	4	5	6
<b>Causes:</b>						
1 (male)	1	1	1	0	0	0
2 (female)	0	0	0	1	1	1
3 (<25)	1	0	0	0	1	0
4 (>=25 and < 65)	0	1	0	0	0	1
5 (>= 65)	0	0	1	1	0	0
<b>Effects:</b>						
100 (Premium is \$1000)	0	1	0	0	0	0
101 (Premium is \$3000)	1	0	0	0	0	0
102 (Premium is \$1500)	0	0	1	1	0	0
103 (Premium is \$500)	0	0	0	0	1	1

- Columns to rows
- Read off the 1's

Test Case #	Inputs (Causes)		Expected Output (Effects)
	Sex	Age	Premium
1	Male	<25	\$3000
2	Male	>=25 and < 65	\$1000
3	Male	>= 65	\$1500
4	Female	>= 65	\$1500
5	Female	<25	\$500
6	Female	>=25 and < 65	\$500

# Notes

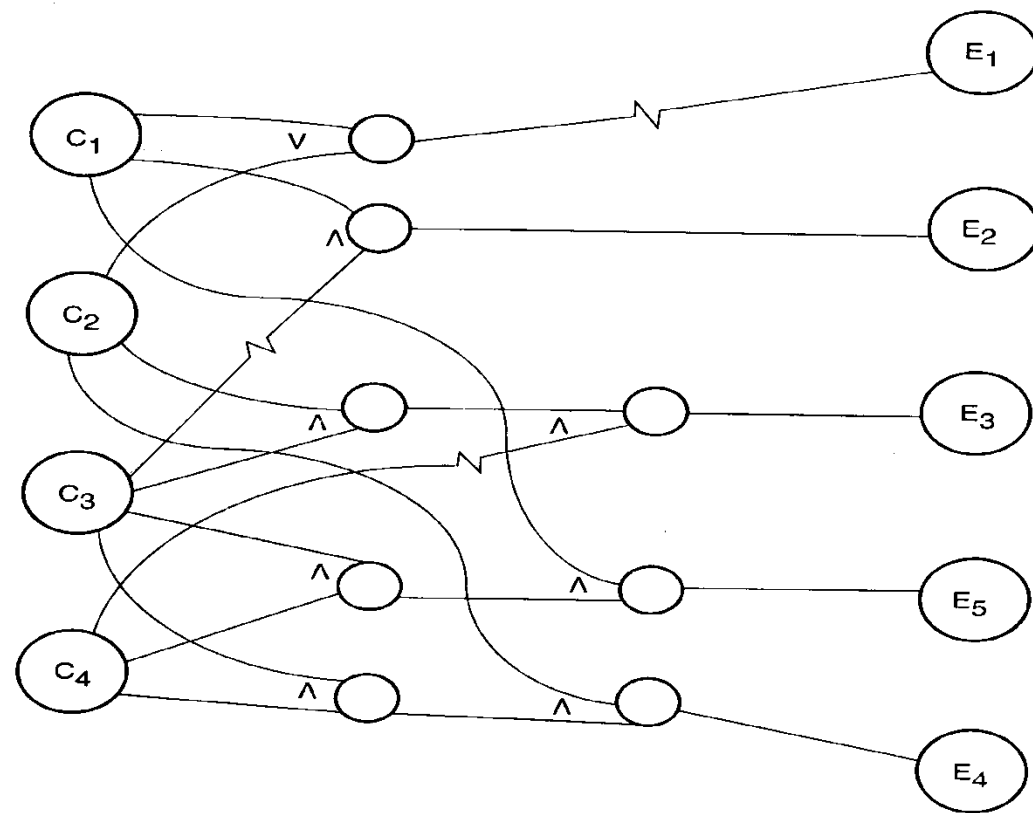
- This was a simple example!
- Good tester could have jumped straight to the end results
- Not always the case....

# Exercise: You try it!

- A bank database which allows two commands
  - Credit acc# amt
  - Debit acc# amt
- Requirements
  - If credit and acc# valid, then credit
  - If debit and acc# valid and amt less than balance, then debit
  - Invalid command – message
- Your task...
  - Identify and name causes and effects
  - Draw CE graphs and add constraints
  - Construct limited entry decision table
  - Construct test cases

# Example...

- Causes
  - C1: command is credit
  - C2: command is debit
  - C3: acc# is valid
  - C4: amt is valid
- Effects
  - Print "Invalid command"
  - Print "Invalid acct#"
  - Print "Debit amt not valid"
  - Debit account
  - Credit account



#	1	2	3	4	5
C1	0	1	x	x	x
C2	0	x	1	1	x
C3	x	0	1	1	1
C4	x	x	0	1	1
E1	1				
E2		1			
E3			1		
E4				1	
E5					1

Testing

# Pair-wise testing

- Often many parameters determine the behavior of a software system
- The parameters may be inputs or settings, and take different values (or different value ranges)
- Many defects involve one condition (single-mode fault), eg. software not being able to print on some type of printer
  - Single mode faults can be detected by testing for different values of different parameters
  - If  $n$  parameters and each can take  $m$  values, we can test for one different value for each parameter 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 often 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 minutes, total 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
  - 1<sup>st</sup> parm will have  $m*m$  with  $n-1$  others
  - 2<sup>nd</sup> parm will have  $m*m$  pairs with  $n-2$
  - 3<sup>rd</sup> 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  $(n-1)+(n-2)+\dots=n(n-1)/2$  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 # of pair wise combinations: 27
- # 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



# 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

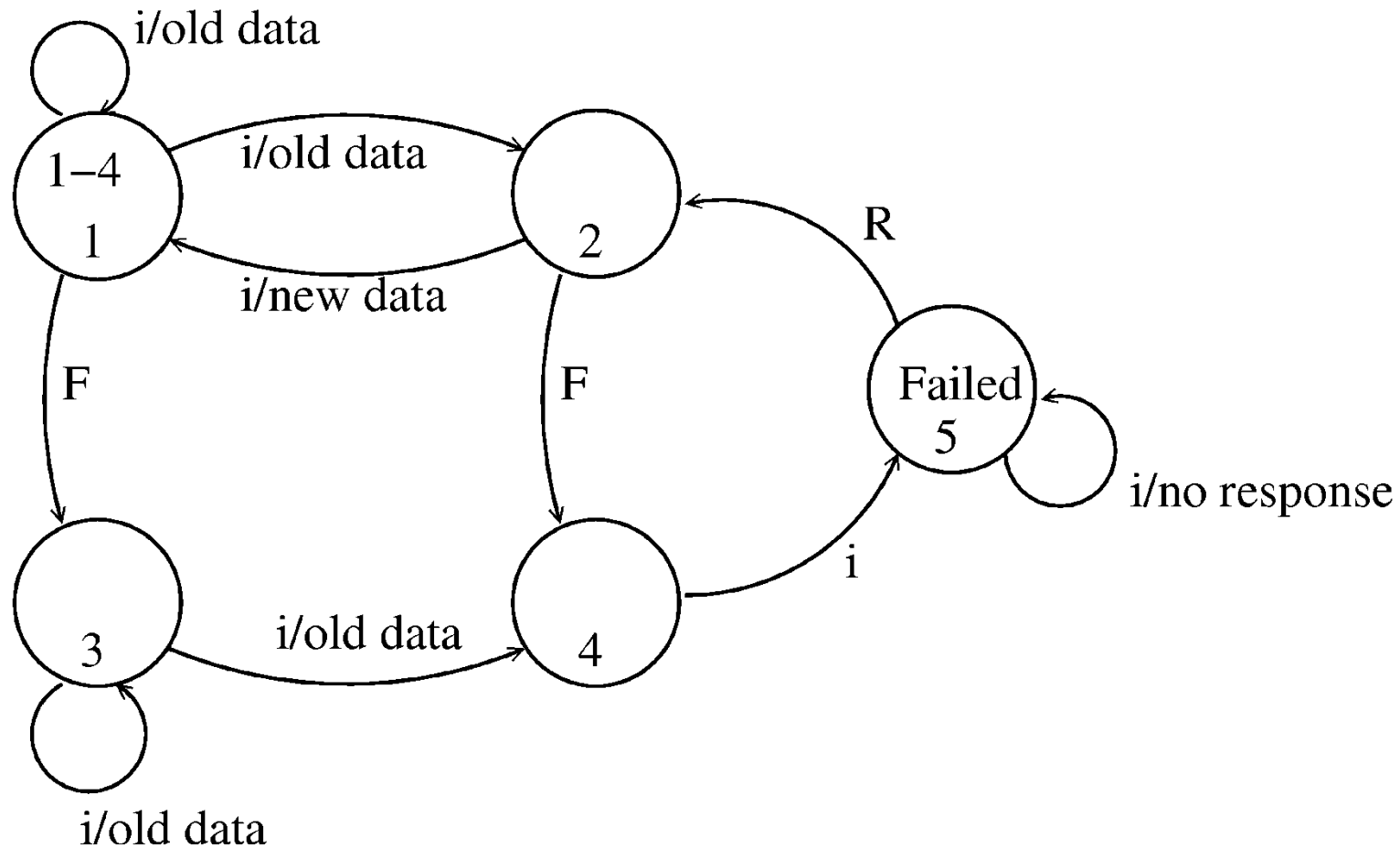
# State-based Testing, example...

- Consider a student survey example
  - A system to take survey of students
  - Student submits survey and is returned results of the survey so far
  - The result may be from the cache (if the database is down) and can be up to 5 surveys old

# State-based Testing, example...

- In a series of requests, first 5 may be treated differently
- Hence, we have two states: one for req no 1-4 (state 1), and other for 5 (2)
- The db can be up or down, and it can go down in any of the two states (3-4)
- Once db is down, the system may get into failed state (5), from where it may recover

# State-based Testing, example...





# 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-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)

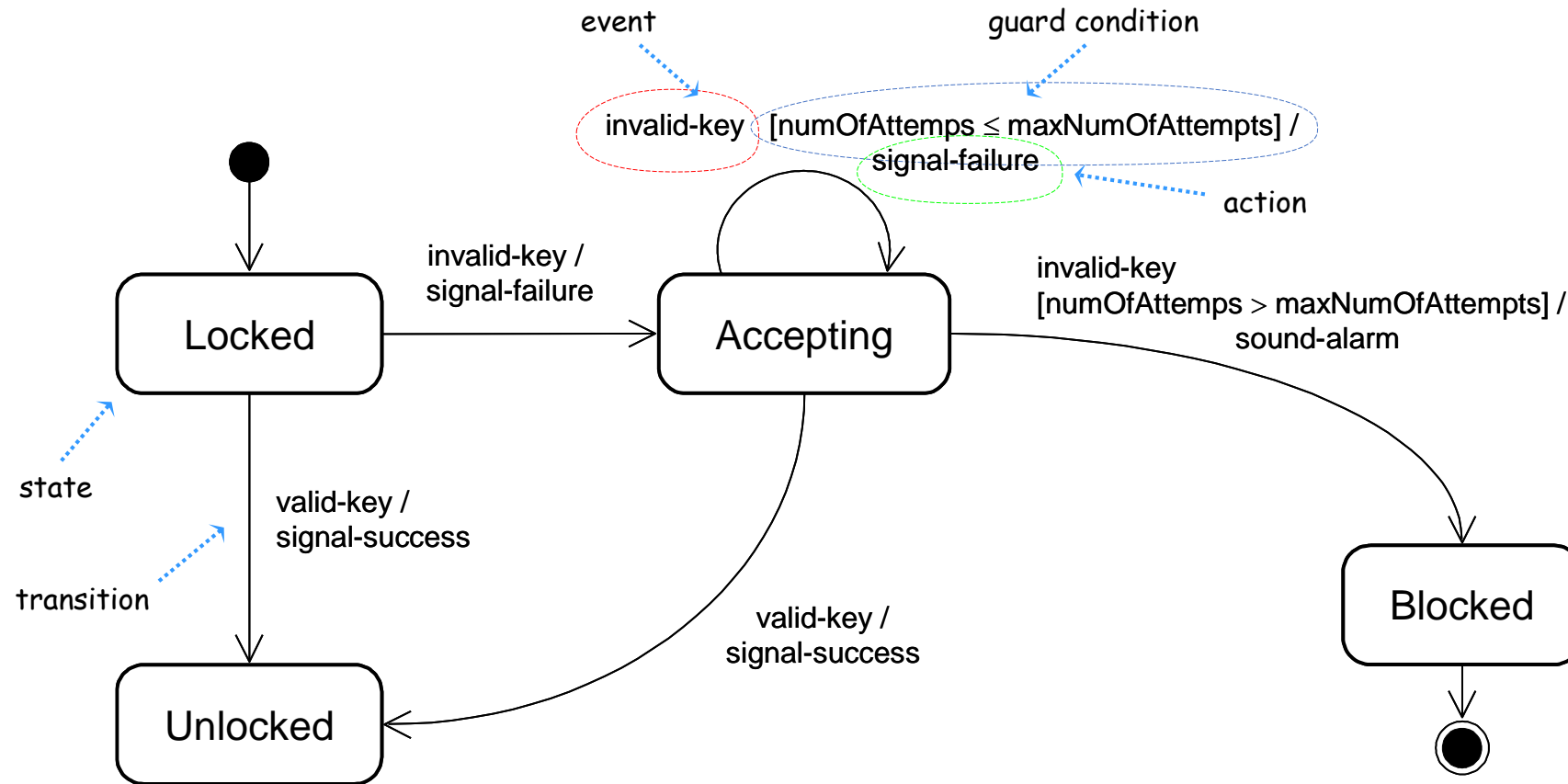
## Example, test cases for AT criteria

SNo	Transition	Test case
1	1 -> 2	Req()
2	1 -> 2	Req(); req(); req(); req();req(); req()
3	2 -> 1	Seq for 2; req()
4	1 -> 3	Req(); fail()
5	3 -> 3	Req(); fail(); req()
6	3 -> 4	Req(); fail(); req(); req(); req();req(); req()
7	4 -> 5	Seq for 6; req()
8	5 -> 2	Seq for 6; req(); recover()

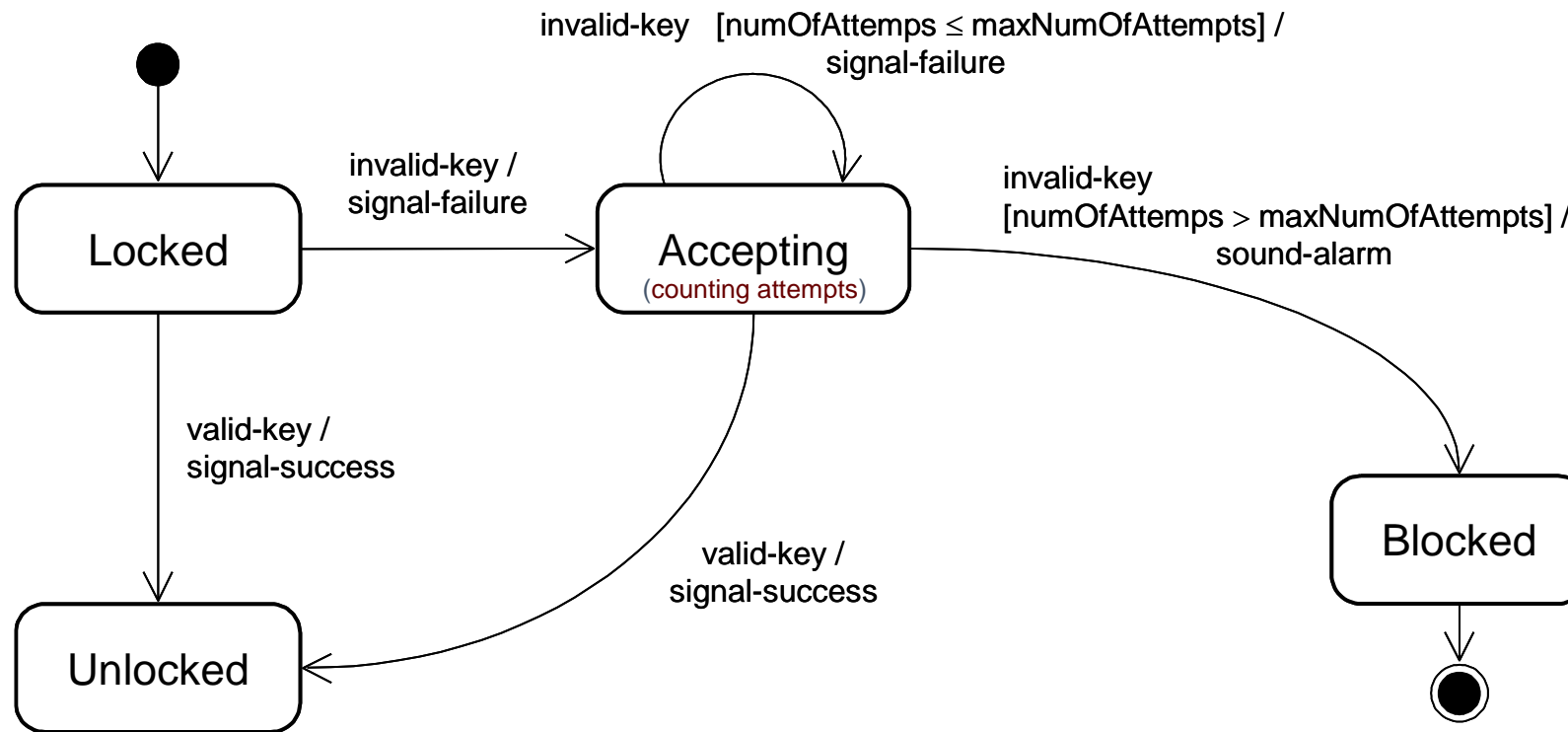
# State-based testing...

- 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)

# State-based Testing Example



# State-based Testing Example



# Controller State Diagram Elements

- Four states  
{ Locked, Unlocked, Accepting, Blocked }
- Two events  
{ valid-key, invalid-key }
- Five valid transitions  
{ Locked→Unlocked, Locked→Accepting,  
Accepting→Accepting, Accepting→Unlocked,  
Accepting→Blocked }

# Ensure State Coverage Conditions

- Cover all identified states at least once  
(each state is part of at least one test case)
- Cover all valid transitions at least once
- Trigger all invalid transitions at least once