SWE 4603

Software Testing and Quality Assurance

Lecture 3

Prepared By Maliha Noushin Raida, Lecturer, CSE
Islamic University of Technology

Lesson Outcome

- Boundary value analysis method
- Equivalence class testing method
- State table-based testing method
- Decision table-based testing method
- Cause-effect graphing method
- Error guessing

Week On:

- Chapter 4: Dynamic Testing: Black-Box Testing Techniques

Testing Categories & Techniques

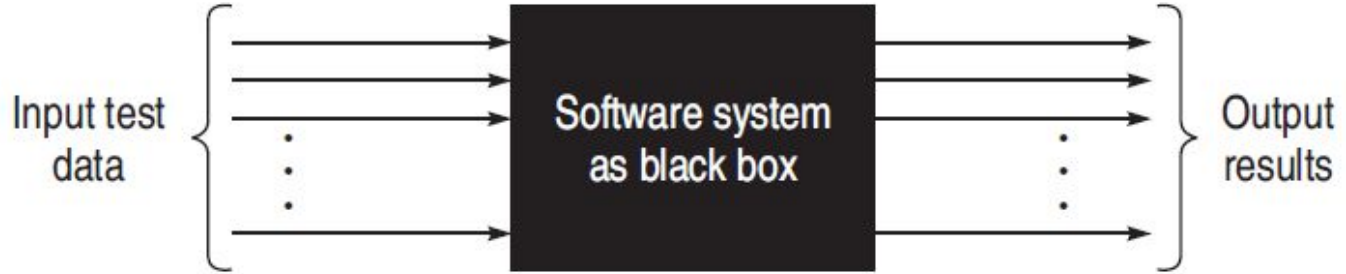
Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

Dynamic Testing

Black-BOx Testing Techniques

Black-Box Testing

- A “black-box” means that one cannot see the internals of the system or inside the box.
- A test or a testing activity that is considered “black-box”, only checks that given some input to the SW system, there is expected output.
- This technique considers only the functional requirements of the software or module.



- An example of such a test is User Acceptance Testing (UAT) where the person doing the testing has no access to the code, is not concerned how the system was implemented, but rather is testing that the system meets the user requirements and works as expected for the typical input that the system needs to work with.

Look at an example

Let’s assume that we are building a small program that employees at a company can use to determine how much their insurance will cost. Here are the requirements for the application.

Requirement #1 An employee must have health insurance to get vision or dental insurance.

Requirement #2 There are two types of health insurance: HMO costs $100 a month; PPO costs $150 a month.

Requirement #3 Vision costs $25 a month

Requirement #4 Dental costs $20 a month

Look at an example

The user interface might look like this. The user selects a health insurance plan and might check boxes to get vision or dental insurance. Then the user clicks the Calculate button. The calculated cost appears in the text field beside the label “Total Cost”.

Black-Box Testing

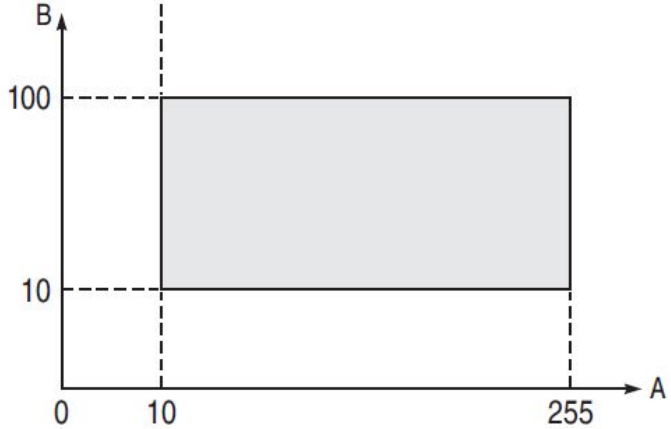
- Black-box testing attempts to find errors in the following categories:
  - To test the modules independently
  - To test the functional validity of the software so that incorrect or missing functions can be recognized
  - To look for interface errors
  - To test the system behavior and check its performance
  - To test the maximum load or stress on the system
  - To test the software such that the user/customer accepts the system within defined acceptable limits

Black-Box Testing: Techniques

1. **Boundary Value Analysis (BVA)**
2. **Equivalence Class Testing**
3. **State Table-Based Testing**
4. **Decision Table-Based Testing**
5. **Cause-Effect Graphing Based Testing**
6. **Error Guessing**

1. Boundary Value Analysis(BVA)

- Test cases designed with boundary input values have a high chance to find errors.
- It means that most of the failures crop up due to boundary values.
- Boundary means the maximum or minimum value taken by the input domain.
- For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101)



Boundary Value Analysis(BVA)

BVA offers several methods to design test cases:

- ✔ Boundary Value Checking (BVC)
- ✔ Robustness Testing Method
- ✔ Worst-case Testing Method

BVA: Boundary Value Checking

- The test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.
- The variable at its extreme value can be selected at:
    1. Minimum value (Min)
    2. Value just above the minimum value (Min+)
    3. Maximum value (Max)
    4. Value just below the maximum value (Max−).

BVA: Boundary Value Checking

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

It can be generalized that for ***n*** variables in a module, ***4n* + 1** test cases can be designed with boundary value checking method.

BVA: Robustness Testing Method

- It is an extension of BVC such that boundary values are exceeded as:
- The variable at its extreme value can be selected at:
    1. A value just greater than the Maximum value **(Max+)**
    2. A value just less than Minimum value **(Min−)**
- So if we consider the previous example then we can add following cases :

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

It can be generalized that for ***n*** variables in a module, ***6n* + 1** test cases can be designed with Robustness Testing method.

BVA: Worst-Case Testing Method

- We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called worst-case testing method.

So if we consider the previous example then we can add following cases to the list of 9 test cases designed in BVC as:

10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. A_{max-}, B_{min}
16. A_{max}, B_{min+}
17. A_{max-}, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, B_{max-}
21. A_{min+}, B_{max-}
22. A_{max}, B_{max}
23. A_{max-}, B_{max}
24. A_{max}, B_{max-}
25. A_{max-}, B_{max-}

It can be generalized that for ***n*** variables in a module, $\mathbf{5^n}$ test cases can be designed with Worst-Case Testing method.

Example

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

A. Test cases using BVC, where n=1
Total # of test case will be 4n+1 = 5

Min value = 1
Min^{+} value = 2
Max value = 100
Max^{-} value = 99
Nominal value = 50–55

min and max values

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Test cases

Example

B. Test cases using Robust testing
Total # of test case will be 6n+1 = 7 [as n=1 here]

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

min and max values

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

Test cases

Example

C. Test cases using worst-case testing
Total # of test case will be $5^n = 5$ [as n=1 here]

So, the number of test cases will be same as BVC.

Example 2

A program computes ab where a lies in the range [1,10] and b within [1,5].Design test cases for this program using **BVC**, **robust testing**, and **worst-case testing** methods.

Solution:

A. Test cases using BVC, as n=2
Total # of test case will be 4n+1 = 9

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

min and max values

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

Test cases

Example 2

B. Test cases using Robust testing: [as n=2]
Total # of test case will be 6n+1 = 13

	a	b
Min value	1	1
Min^- value	0	0
Min^+ value	2	2
Max value	10	5
Max^+ value	11	6
Max^- value	9	4
Nominal value	5	3

min and max values

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

Test cases

Example 2

C. Test cases using worst-case testing
Total # of test case will be $5^n = 25$ [as n=2 here]

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

min and max values

Example 2

Worst-case testing
Test cases

Test Case ID	a	b	Expected Output
1	1	3	3
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125
10	1	1	1
11	1	2	1
12	1	5	1
13	1	4	1
14	10	1	10
15	10	2	100
16	10	5	100000
17	10	4	10000
18	2	1	2
19	2	2	4
20	2	5	32
21	2	4	16
22	9	1	9
23	9	2	81
24	9	5	59049
25	9	4	6561

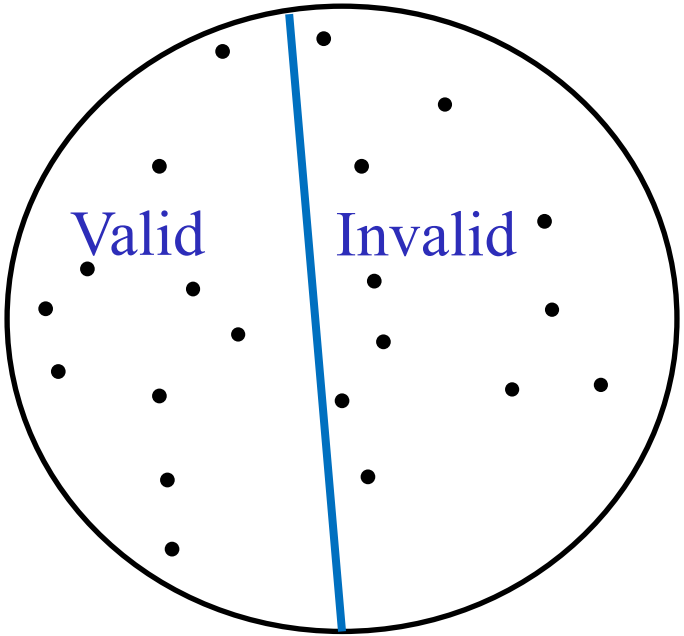
2. Equivalent-Class Testing

- Typically the universe of all possible test cases is so large that you cannot try them all
- You have to select a relatively small number of test cases to actually run
- Which test cases should you choose?
- Equivalence partitioning helps answer this question
- Partition the test cases into "equivalence classes"
- Each equivalence class contains a set of "equivalent test cases”
- Two test cases are considered to be equivalent if we expect the program to process them both in the same way (i.e., follow the same path through the code)
- If you expect the program to process two test cases in the same way, only test one of them, thus reducing the number of test cases you have to run

Equivalent-Class Testing

Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid test cases



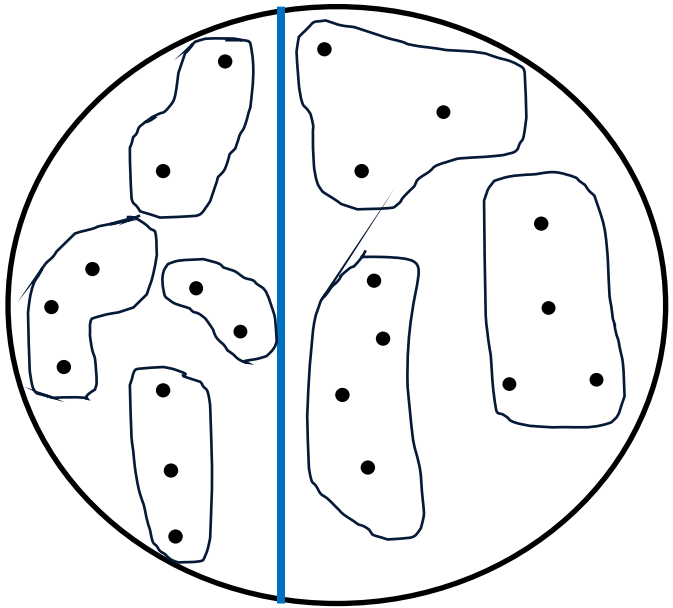
Valid equivalence classes: These classes consider valid inputs to the program.

Invalid equivalence classes: We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program

Equivalent-Class Testing

Equivalence Partitioning

- Partition valid and invalid test cases into equivalence classes

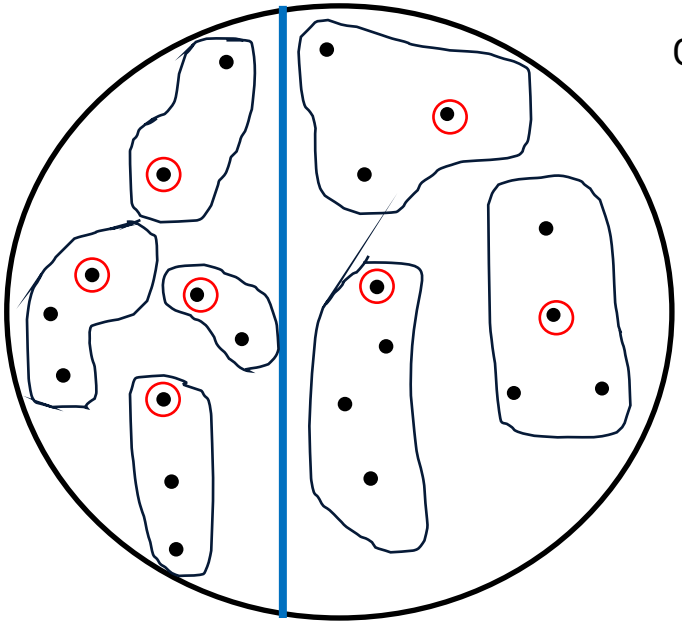


There are no well-defined rules for identifying equivalence classes, as it is a heuristic process.

Equivalent-Class Testing

Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



Guidelines are given below to identify test cases:

- Assign a unique identification number to each equivalence class
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases
- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases

Equivalent-Class Testing(EC)

There are 4 sub-techniques of Equivalence Class Testing.

1. Weak Normal Testing :WN
2. Strong Normal Testing :SN
3. Weak Robust Testing :WR
4. Strong Robust Testing :SR

Equivalent-Class Testing(EC)

- Function F is implemented and a function F, of two variables x1 and x2.
- x1 and x2 have the following boundaries and intervals within boundaries:
  - a=<x1=<d with intervals [a,b), [b,c), [c,d]
  - e=<x2=<g with intervals [e,f), [f,g]
- So, invalid value for x1 and x2 as follows,
  - x1 < a and x1 > d
  - x2 < e and x2 > g

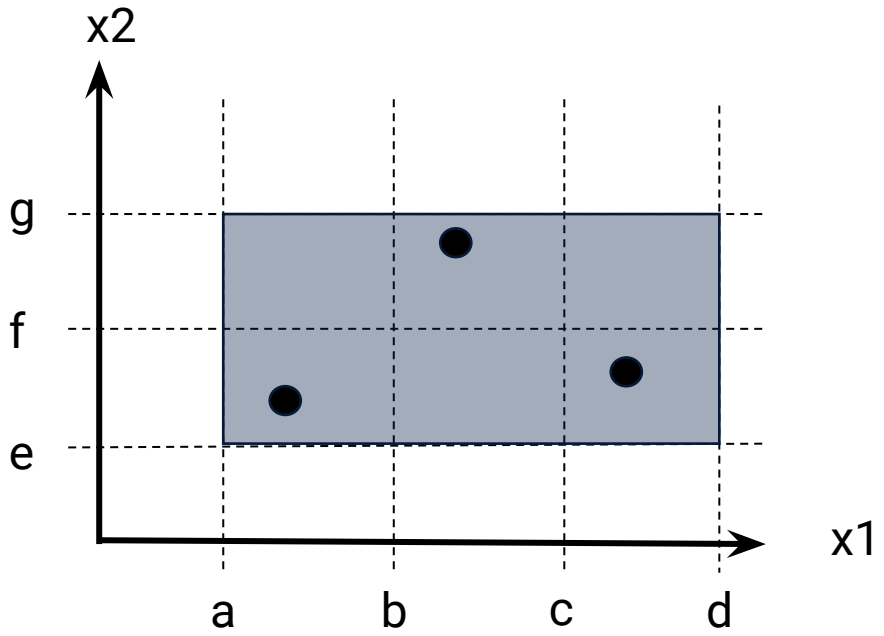
Remarks: [= closed interval, (= open interval

Equivalent-Class Testing(EC)

1. Weak Normal Testing :WN

Valid EC:

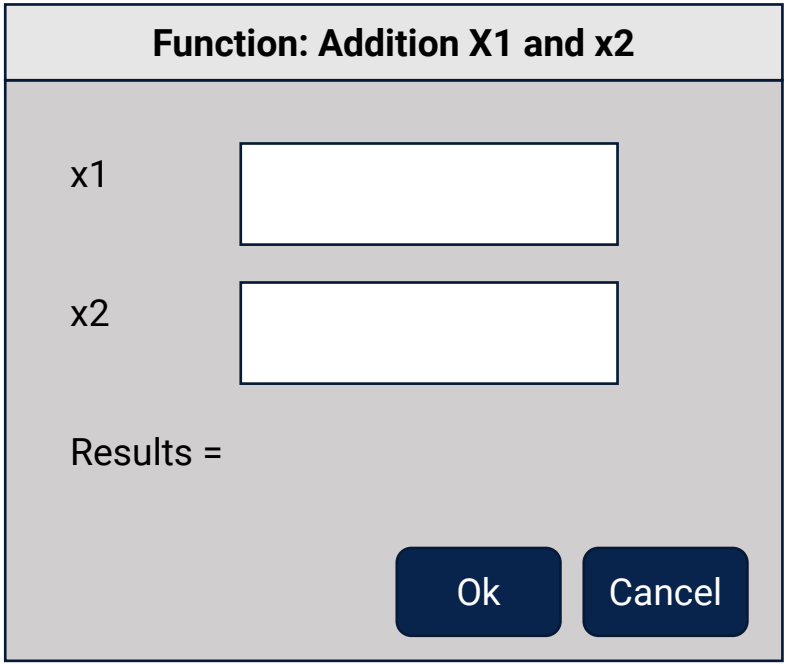
$$Ec1 = \{x1 : a \leq x1 < b\}$$
$$Ec2 = \{x1 : b \leq x1 < c\}$$
$$Ec3 = \{x1 : c \leq x1 \leq d\}$$
$$Ec4 = \{x2 : e \leq x2 < f\}$$
$$Ec5 = \{x2 : f \leq x2 \leq g\}$$



One variable from each equivalence class is tested by the team. Weak normal equivalence class testing is also known as **single fault assumption**.

Equivalent-Class Testing(EC)

Example of Weak-Normal testing: Addition of x1 and x2



Equivalent-Class Testing(EC)

Example of Weak-Normal testing:

Valid EC:

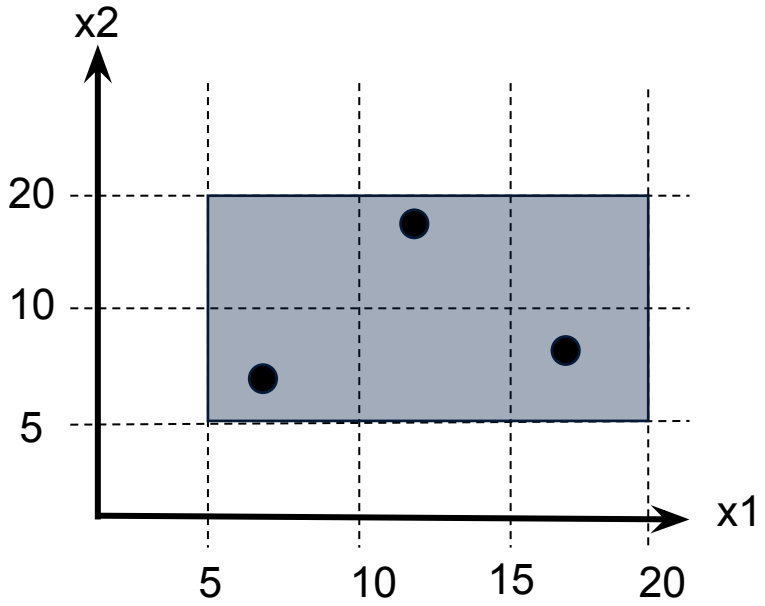
Ec1 = {x1: 5=<x1< 10}

Ec2= {x1: 10=<x1< 15}

Ec3 = {x1: 15 <= x1 <= 20}

Ec4 = {x2: 5 =<x2 <10}

Ec5 = {x2: 10=< x2 <=20}

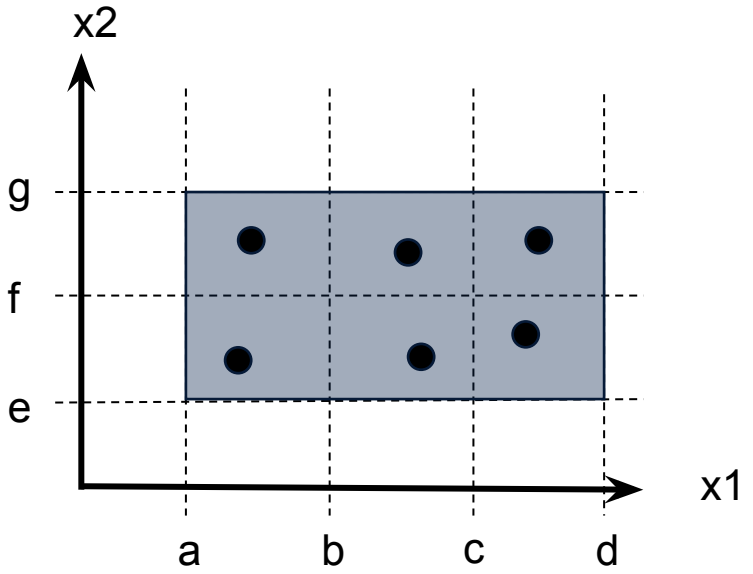


Test case ID	EC ID	x1	x2	Expected Results
WN1	EC1, EC4	5	6	11
WN2	EC2, EC5	12	15	27
WN3	EC3, EC4	17	7	24

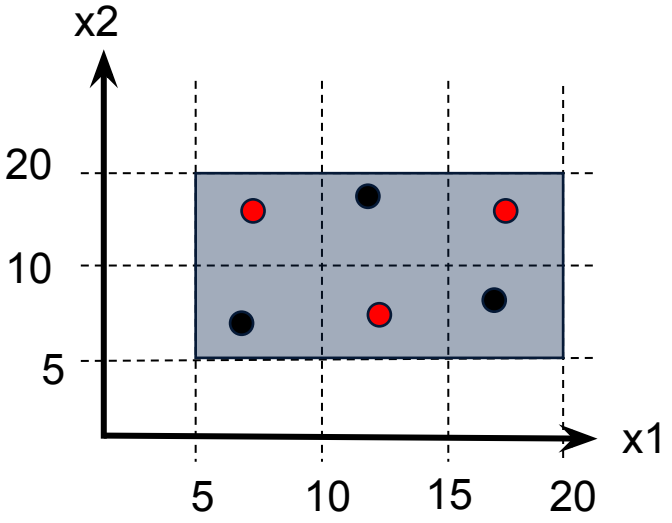
Equivalent-Class Testing(EC)

2. Strong Normal Testing : SN

- Test cases taken from each element of Cartesian product of the equivalence classes. Cartesian product guarantees notion of completeness.
- SN is a “**multiple fault assumption**”



Equivalent-Class Testing(EC)



Example of Strong-Normal testing:

Valid EC:

Ec1 = {x1: 5=<x1< 10}

Ec2= {x1: 10=<x1< 15}

Ec3 = {x1: 15 <= x1 <= 20}

Ec4 = {x2: 5 =<x2 <10}

Ec5 = {x2: 10=< x2 <=20}

Test case ID	EC ID	x1	x2	Expected Results
SN1	EC1, EC4	5	6	11
SN2	EC1, EC5	7	12	19
SN3	EC2, EC4	10	5	15
SN4	EC2, EC5	12	15	27
SN5	EC3, EC4	17	7	24
SN6	EC3, EC5	15	15	30

Equivalent-Class Testing(EC)

3. Weak Robust Testing :WR

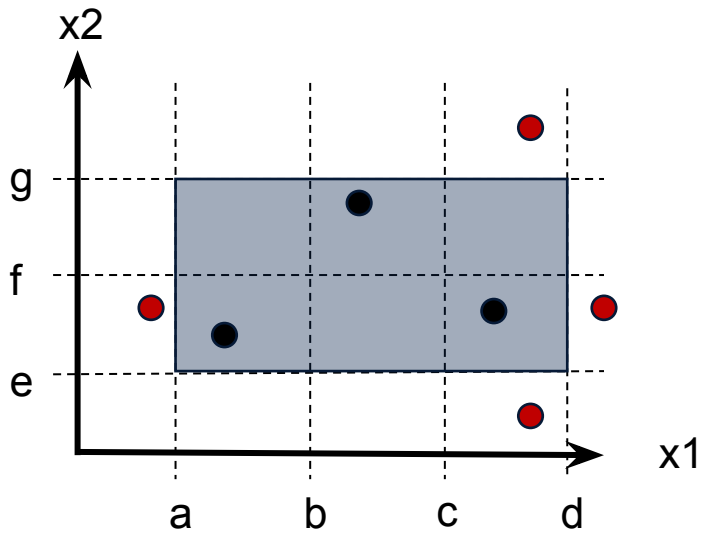
Additional consider in Invalid EC:

Ec6 = {x1: x1 < a}

Ec7 = {x1: x1 > d}

Ec8 = {x2 : x2 < e}

Ec9 = {x2 : x2 > g}



- Robust - consideration of invalid values and extension to WN.
- Invalid inputs – each test case has one invalid value, single fault should cause failure as “single fault assumption”.

Equivalent-Class Testing(EC)

Example of Weak-Robust testing:

Invalid EC:

Ec6 = {x1: x1< 5}

Ec7= {x1: x1>15}

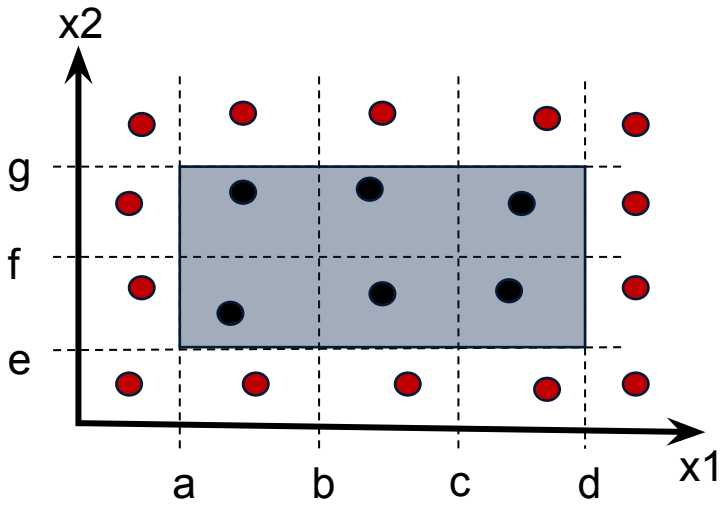
Ec8 = {x2: x2 < 10}

Ec9 = {x2: x2 >20}

Test case ID	EC ID	x1	x2	Expected Results
WR1	EC1, EC4	5	6	11
WR2	EC2, EC5	12	15	27
WR3	EC3, EC4	17	7	24
WR4	EC6, EC4	4	8	X1 is out of range
WR5	EC7, EC4	23	8	X1 is out of range
WR6	EC3, EC8	18	4	X2 is out of range
WR7	EC3, EC9	18	23	X2 is out of range

Equivalent-Class Testing(EC)

4. Strong Robust Testing :SR



- Robust - consideration of invalid values and extension to SN.
- Strong – multiple faults assumption.
- Test cases taken from each element of Cartesian product of the Valid EC and Invalid EC

Equivalent-Class Testing(EC)

Next Date Problem:

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

- $1 \leq mm \leq 12$
- $1 \leq dd \leq 31$
- $1900 \leq yyyy \leq 2025$
- Its output would be the next date or an error message ‘invalid date.’

Design test cases using equivalence class partitioning method.

Equivalent-Class Testing(EC)

Next Date Problem:

- Valid EC
  - M1 = {month: 1 =< month =<12}
  - D1 = {day: 1 =< day =< 31}
  - Y1 = {year: 1900 =< year =< 2025}
- Invalid EC
  - M2 = {month: month <1}
  - M3 = {month: month >12}
  - D2 = {day: day <1}
  - D3 = {day: day >31}
  - Y2 = {year: year < 1900}
  - Y3 = {year: year > 2025}

Equivalent-Class Testing(EC)

Next Date Problem: Weak Normal and Strong Normal Test Cases

Test Case	Month	Day	Year	Expected Results
WN1, SN1	6	15	1912	6/16/1912

Equivalent-Class Testing(EC)

Next Date Problem: Weak Robust Test Cases

Test Case ID	Month	Day	Year	Expected Results
WN1	6	15	1912	6/16/1912
WR1	-1	15	1912	invalid date
WR2	13	15	1912	invalid date
WR3	6	-1	1912	invalid date
WR4	6	32	1912	invalid date
WR5	6	15	1811	invalid date
WR6	6	15	2028	invalid date

Equivalent-Class Testing(EC)

Next Date Problem: Strong Robust Test Cases

Test Case	Month	Day	Year	Expected Results
WN1	6	15	1912	6/16/1912
SR1	-1	15	1912	invalid date
SR2	6	-1	1912	invalid date
SR3	6	15	1811	invalid date
SR4	-1	-1	1912	invalid date
SR5	6	-1	1811	invalid date
SR6	-1	15	1811	invalid date
SR7	-1	-1	1811	invalid date

Equivalent-Class Testing(EC)

Improved Input Equivalence Classes: Next Date Problem

- M1 = {month: month has 30 days}
- M2 = {month: month has 31 days}
- M3 = {month: month = February}
- D1 = {day: 1 =< day =< 28}
- D2 = {day: day = 29}
- D3 = {day: day = 30}
- D4 = {day: day = 31}
- Y1 = {year: year is leap year}
- Y2 = {year: year is common year }

State Table Based Testing

- Finite State Machine
  - FSM is a behavioral model whose outcome depends upon both previous and current inputs.
  - It can be in exactly one of a finite number of states at any given time.
- State Transition Diagram / State Graph
  - A state graph is the pictorial representation of an FSM.
  - Its purpose is to depict the states that a system or its components can assume.
  - It shows the events or circumstances that cause or result from a change from one state to another

For example, a task in an operating system can have the following states:

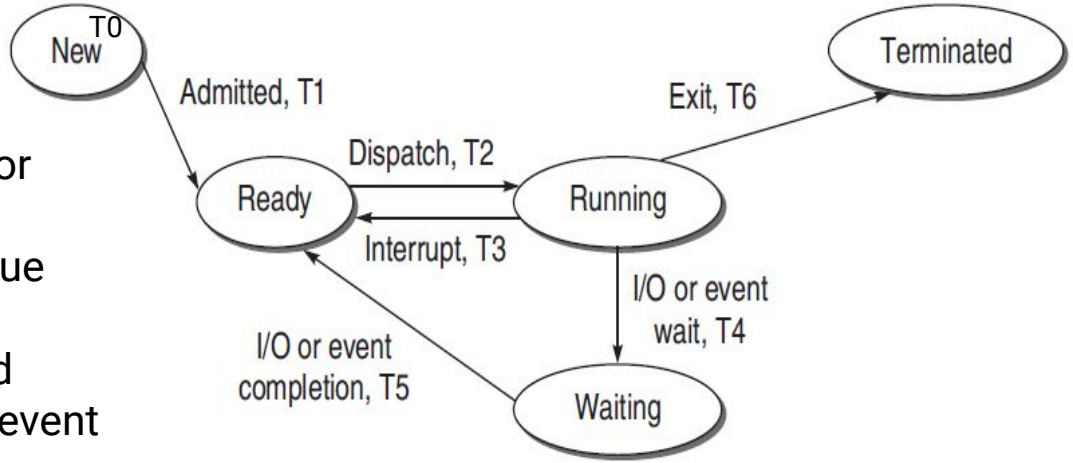
1. **New State**: When a task is newly created.
2. **Ready**: When the task is waiting in the ready queue for its turn.
3. **Running**: When instructions of the task are being executed by CPU.
4. **Waiting**: When the task is waiting for an I/O event or reception of a signal.
5. **Terminated**: The task has finished execution.

State Table Based Testing

The state graph of task states is as below:

Each arrow link provides two types of information:

1. Transition events like admitted, dispatch, interrupt, etc.
2. The resulting output from a state like T1, T2, T3, etc.



T0 = Task is in new state and waiting for admission to ready queue
T1 = A new task admitted to ready queue
T2 = A ready task has started running
T3 = Running task has been interrupted
T4 = Running task is waiting for I/O or event
T5 = Wait period of waiting task is over
T6 = Task has completed execution

State Table Based Testing

- State Graphs are complex for large systems.
- State Graphs are converted to tabular format known as state table.
- following conventions are used for state table :
  - Each row of the table corresponds to a state.
  - Each column corresponds to an input condition.
  - The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

State Table Based Testing

- Now, we can start testing with state tables.
- procedure for converting state graphs and state tables into test cases is as below:
  - **Identify the states**: The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the database. **Example:** the state could be composed of the values of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of 2 × 2 × 10 = 40 states.
  - **Prepare state transition diagram after understanding transitions between states**: After having all the states, identify the inputs on each state and transitions between states and prepare the state graph. Every input state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state.
  - **Convert the state graph into the state table as discussed earlier**
  - **Analyze the state table for its completeness**

State Table Based Testing

- **Create the corresponding test cases from the state table**
  - Test cases are produced in a tabular form known as the test case table which contains six columns as shown below
    - Test case ID : a unique identifier for each test case
    - Test Source : a trace back to the corresponding cell in the state table
    - Current State : the initial condition to run the test
    - Event : the input triggered by the user
    - Output : the current value returned
    - Next State : the new state achieved

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State

State Table Based Testing

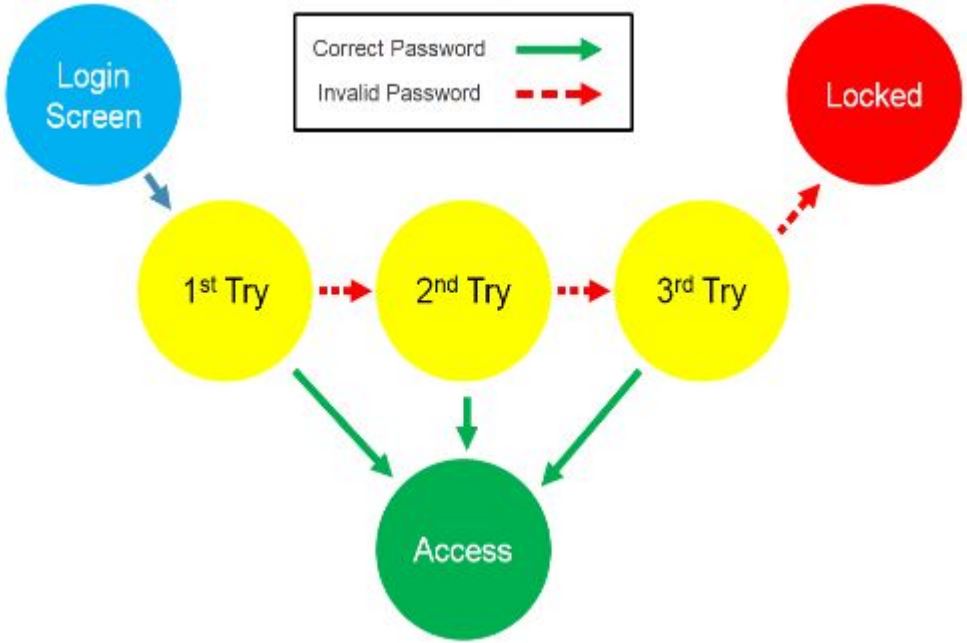
Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready

Test case TC1 gets information from Cell 1 and indicates that one task is created ***New*** in the system. This new task is getting the input event ***Admit*** and transitions to the new state ***Ready*** and the output is **T1**

State Table Based Testing

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

State Table Based Testing



Decision Table-Based Testing

- Decision table is another useful method to represent the information in a tabular method.
- It has the specialty to consider complex combinations of input conditions and resulting actions.

Formation of decision Table:

		Rule 1	Rule 2	Rule 3	Rule 4	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Condition stub It is a list of input conditions for which the complex combination is made.
Action stub It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

Decision Table-Based Testing

		Rule 1	Rule 2	Rule 3	Rule 4	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Condition entry It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE

Action entry It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

Decision Table-Based Testing

Guidelines to develop a decision table:

- List all actions that can be associated with a specific procedure (or module).
- List all conditions (or decision made) during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

Decision Table-Based Testing

For designing test cases from a decision table, following interpretations should be done:

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the test case itself.
- If there are k rules over n binary conditions, there are at least k test cases and at the most 2n test cases.

Decision Table-Based Testing

Example:
A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

F- False
T- True
I- Don't Case

		ENTRY		
		Rule 1	**Rule 2**	**Rule3**
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

Decision Table-Based Testing

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Decision Table-Based Testing

Example:

A university is admitting students in a professional course subject to the following conditions:

(a) Marks in Java ≥ 70
(b) Marks in C++ ≥ 60
(c) Marks in OOAD ≥ 60
(d) Total in all three subjects ≥ 220 OR Total in Java and C++ ≥ 150

Also, If the aggregate mark of an eligible candidate is more than 240, he will be eligible for scholarship course, otherwise he will be eligible for normal course.

The program reads the marks in the three subjects and generates the following outputs:

(i) Not eligible
(ii) Eligible for scholarship course
(iii) Eligible for normal course

Design test cases for this program using decision table testing.

Decision Table-Based Testing

Solution:

	ENTRY									
	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**
C1: marks in Java ≥ 70	T	T	T	T	F	I	I	I	T	T
C2: marks in C++ ≥ 60	T	T	T	T	I	F	I	I	T	T
C3: marks in OOAD ≥ 60	T	T	T	T	I	I	F	I	T	T
C4:Total in three subjects ≥ 220	T	F	T	T	I	I	I	F	T	T
C5:Total in Java & C++ ≥ 150	F	T	F	T	I	I	I	F	T	T
C6:Aggregate marks > 240	F	F	T	T	I	I	I	I	F	T
A1: Eligible for normal course	X	X							X	
A2: Eligible for scholarship course			X	X						X
A3: Not eligible					X	X	X	X		

Decision Table-Based Testing

The test cases derived from the decision table are given below:

Test Case ID	Java	C++	OOAD	Aggregate Marks	Expected Output
1	70	75	60	224	Eligible for normal course
2	75	75	70	220	Eligible for normal course
3	75	74	91	242	Eligible for scholarship course
4	76	77	89	242	Eligible for scholarship course
5	68	78	80	226	Not eligible
6	78	45	78	201	Not eligible
7	80	80	50	210	Not eligible
8	70	72	70	212	Not eligible
9	75	75	70	220	Eligible for normal course
10	76	80	85	241	Eligible for scholarship course

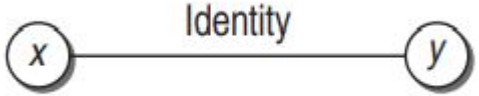
Cause-Effect Graphing Based Testing

- Like decision tables, cause-effect graphing is another technique for combinations of input conditions..
- Here causes are the input conditions and effects are the results of those input conditions.
- starts with a set of requirements and determines the minimum possible test cases for maximum test coverage which reduces test execution time and cost.
- The goal is to reduce the total number of test cases, still achieving the desired application quality by covering the necessary test cases for maximum coverage.
- A cause is a distinct input condition identified in the problem. Similarly, an effect is an output condition

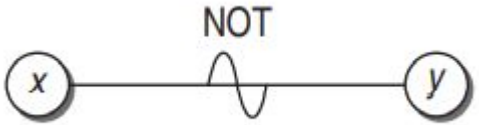
Cause-Effect Graphing Based Testing

Common Notations for Cause Effect Graph:

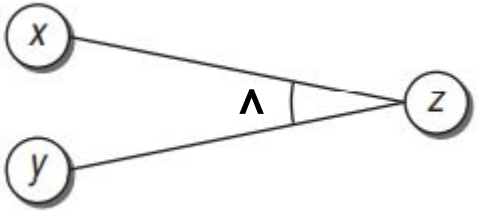
Identity: if x is true, y is true; else y is false.



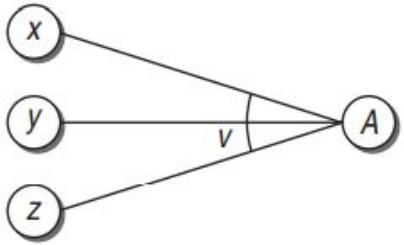
NOT: if x is true, y is false; else y is true.



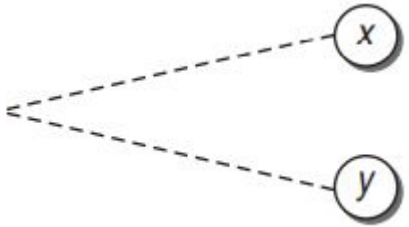
AND: if both x and y are true, z is true; else z is false.



OR: if x or y or z is true, A is true; else A is false.



Mutually Exclusive: One and only one of x and y must be 1,



Cause-Effect Graphing Based Testing

Example:

- The “Print message” is software that reads two characters and, depending on their values, messages is printed.
  - The first character must be an “A” or a “B”.
  - The second character must be a digit.
  - If the first character is an “A” or “B” and the second character is a digit, then the file must be updated.
  - If the first character is incorrect (not an “A” or “B”), the message X must be printed.
  - If the second character is incorrect (not a digit), the message Y must be printed.

Cause-Effect Graphing Based Testing

Solution:

The Causes of this situation are:
C1 – First character is A
C2 – First character is B
C3 – the Second character is a digit
The Effects (results) for this situation are:
E1 – Update the file
E2 – Print message “X”
E3 – Print message “Y”

Cause-Effect Graphing Based Testing

Try to figure out:

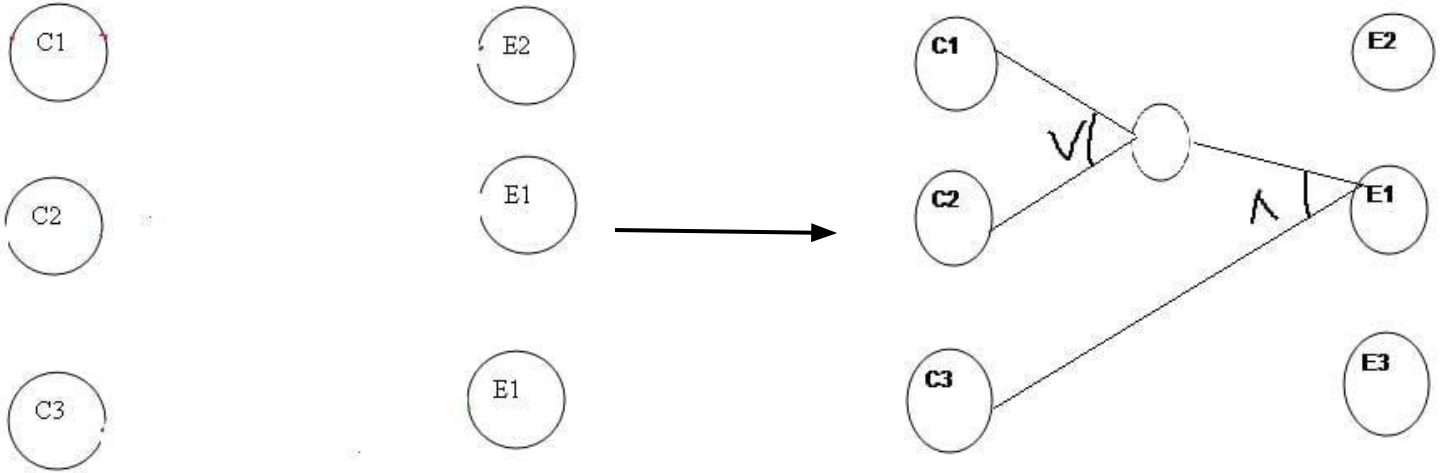
In this example, let's start with Effect E1.
Effect E1 is for updating the file. The file is updated when:

- The first character is "A" and the second character is a digit
- The first character is "B" and the second character is a digit
- The first character can either be "A" or "B" and cannot be both.

Now let's put these 3 points in symbolic form:
For E1 to be true – the following are the causes:

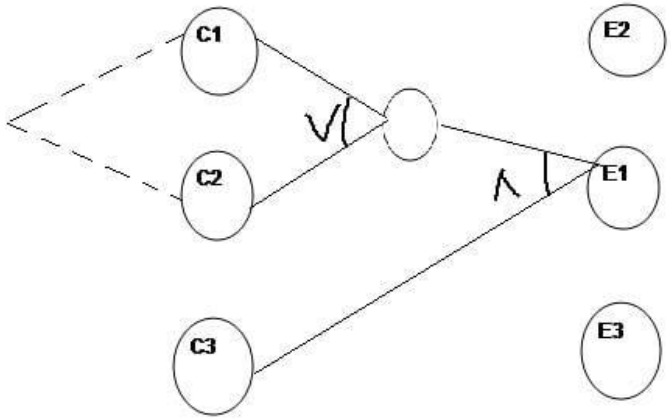
- C1 and C3 should be true
- C2 and C3 should be true
- C1 and C2 cannot be true together. This means C1 and C2 are mutually exclusive.

Cause-Effect Graphing Based Testing



for E1 to be true the condition is $(C1 \bigvee C2) \bigwedge C3$

Cause-Effect Graphing Based Testing

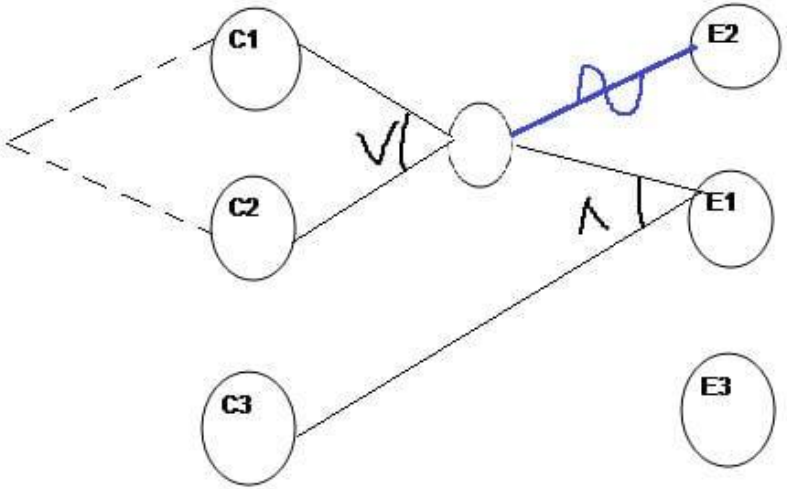


There is a third condition where C1 and C2 are mutually exclusive. So the final graph for effect E1 to be true is shown below:

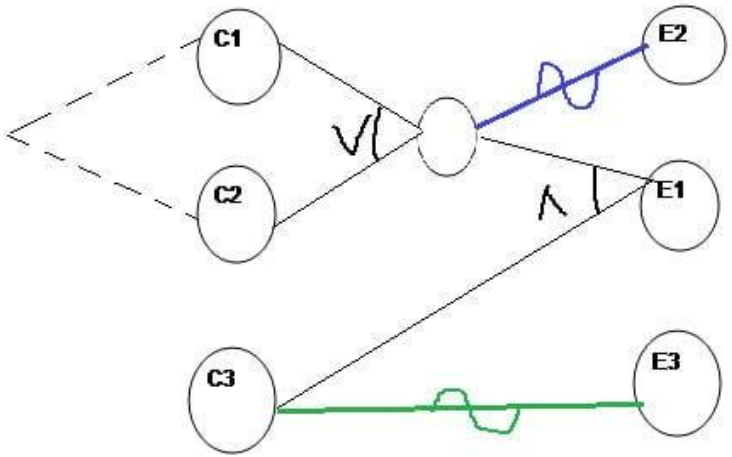
Cause-Effect Graphing Based Testing

E2 states print message “X”. Message X will be printed when the First character is neither A nor B.

This means Effect E2 will hold true when either C1 OR C2 is invalid. So the graph for Effect E2 is shown as (In blue line)



Cause-Effect Graphing Based Testing



E3 states print message “Y”. Message Y will be printed when the Second character is incorrect.

This means Effect E3 will hold true when C3 is invalid. So the graph for Effect E3 is shown as (In Green line)

Cause-Effect Graphing Based Testing

	R1	R2	R3	R4	R5	R6
C1	T		F		T	
C2		T		F		T
C3	T	T	F	T	F	F
E1	X	X				
E2			X	X		
E3					X	X

TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

Error Guessing

- Error guessing is the preferred method used when all other methods fail.
- According to this method, errors or bugs can be guessed which do not fit in any of the earlier defined situations..
- So test cases are generated for these special cases.
- this capability comes with years of experience in a particular field of testing.
- error guessing is an ad hoc approach, based on intuition, experience, knowledge of project, and bug history.

Error Guessing

- For example, consider the system for calculating the roots of a quadratic
- equation. Some special cases in this system are as follows:
- What will happen when a = 0? Though, we do consider this case, there are chances that we overlook it while testing, as it has two cases:

  (i) If a = 0 then the equation is no longer quadratic.

  (ii) For calculation of roots, division is by zero.
- What will happen when all the inputs are negative?
- What will happen when the input list is empty?