Software Quality Assurance and Testing Lecture - 04

Dynamic Testing

BLACK BOX TESTING TECHNIQUES

Boundary Value Analysis (BVA)

- An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that 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.

- BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, 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 Checking (BVC)

- In this method, 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:
  - Minimum value (Min)
  - Value just above the minimum value (Min^{+})
  - Maximum value (Max)
  - Value just below the maximum value (Max^{-})

Boundary Value Checking (BVC)

- Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following tests ($4n + 1$) 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}

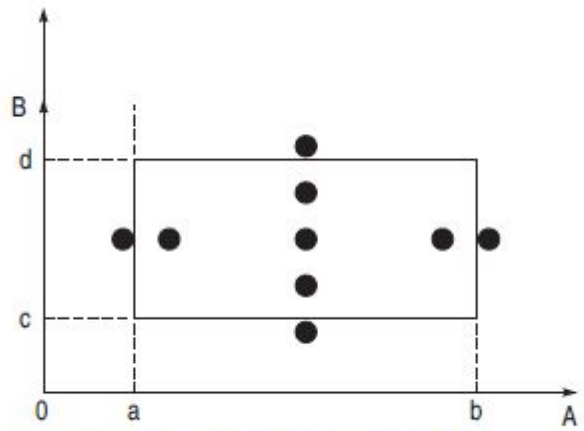


Figure 4.3 Boundary value checking

Robustness Testing Method

- The idea of BVC can be extended such that boundary values are exceeded as:
  - value just greater than the Maximum value (Max^{+})
  - value just less than Minimum value (Min^{-})
- When test cases are designed considering the above points in addition to BVC, it is called robustness testing.
- For n input variables in a module, $6n + 1$ test cases

Robustness Testing Method

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

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. 5^n 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}
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-}

Example

- A program computes a^b 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.

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

BVC

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

Robust Testing

	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

Robust Testing

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

Worst Testing

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

Worst Testing

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

Equivalence Class Testing

- We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or a class of data. Equivalence partitioning is a method for deriving test cases where in classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur.

Equivalence Class Testing

- Equivalence partitioning method for designing test cases has the following goals:
  - **Completeness:** Without executing all the test cases, we strive to touch the completeness of testing domain.
  - **Non-redundancy:** When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

Two steps for equivalence partitioning

- Identify equivalence classes
- Design Test cases

Identification Of Equivalent Classes

- Two types of classes can always be identified :
  - **Valid equivalence classes:** These classes consider valid inputs to the program.
  - **Invalid equivalence classes:** One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behavior of the program

Guidelines for Forming Equivalence Classes

- If there is no reason to believe that the entire range of an input will be treated in the same manner, then the range should be split into two or more equivalence classes.
- If a program handles each valid input differently, then define one valid equivalence class per valid input.
- Boundary value analysis can help in identifying the classes. For example, for an input condition, say $0 \leq a \leq 100$, one valid equivalent class can be formed from the valid range of a. And with BVA, two invalid classes that cross the minimum and maximum values can be identified, i.e. $a < 0$ and $a > 100$.

Guidelines for Forming Equivalence Classes

- If an input variable can identify more than one category, then for each category, we can make equivalent classes. For example, if the input is a character, then it can be an alphabet, a number, or a special character. So we can make three valid classes for this input and one invalid class.
- If an input condition specifies a ‘must be’ situation (e.g., ‘first character of the identifier must be a letter’), identify a valid equivalence class (it is a letter) and an invalid equivalence class (it is not a letter).

Guidelines for Forming Equivalence Classes

- Equivalence classes can be of the output desired in the program. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

Guidelines for Forming Equivalence Classes

- Look for membership of an input condition in a set or group and identify valid (within the set) and invalid (outside the set) classes. For example, if the requirements state that a valid province code is ON, QU, and NB, then identify: the valid class (code is one of ON, QU, NB) and the invalid class (code is not one of ON, QU, NB).

Guidelines for Forming Equivalence Classes

- If the requirements state that a particular input item match a set of values and each case will be dealt with differently, identify a valid equivalence class for each element and only one invalid class for values outside the set. For example, if a discount code must be input as P for a preferred customer, R for a standard reduced rate, or N for none, and if each case is treated differently, identify: the valid class code = P, the valid class code = R, the valid class code = N, the invalid class code is not one of P, R, N.

Guidelines for Forming Equivalence Classes

- If an element of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements. For example, a bank account balance may be from 0 to Rs 10 lakh and balances of Rs 1,000 or more are not subject to service charges. Identify: the valid class: (0 ≤balance <Rs 1,000), i.e. balance is between 0 and Rs 1,000 – not including Rs 1,000; the valid class: (Rs 1,000 ≤ balance ≤Rs 10 lakh, i.e. balance is between Rs 1,000 and Rs 10 lakh inclusive the invalid class: (balance <0) the invalid class: (balance >Rs 10 lakh).

Guidelines for Identifying the Test Cases

- Assign a unique identifi cation 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.

Example

- A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

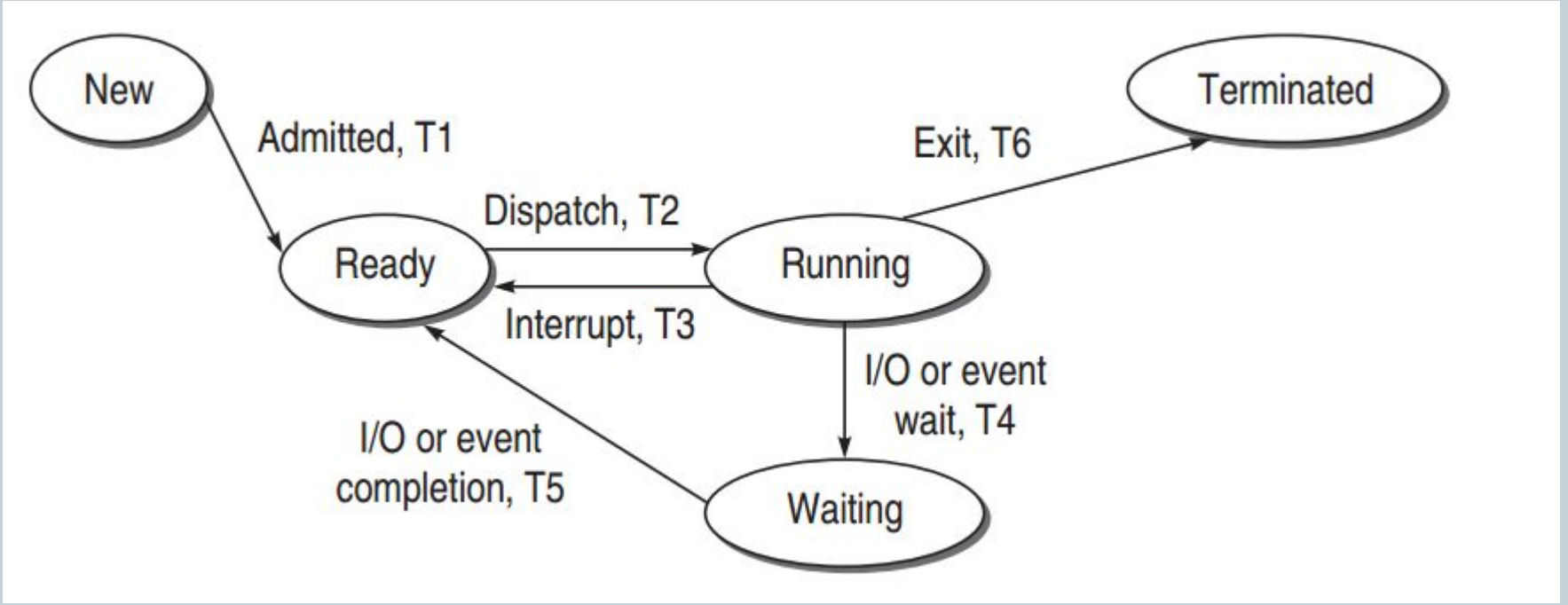
Example

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

State Table-based Testing

- A system or its components may have a number of states depending on its input and time. For example, a task in an operating system can have the following states:
  - New State: When a task is newly created.
  - Ready: When the task is waiting in the ready queue for its turn.
  - Running: When instructions of the task are being executed by CPU.
  - Waiting: When the task is waiting for an I/O event or reception of a signal.
  - Terminated: The task has finished execution.

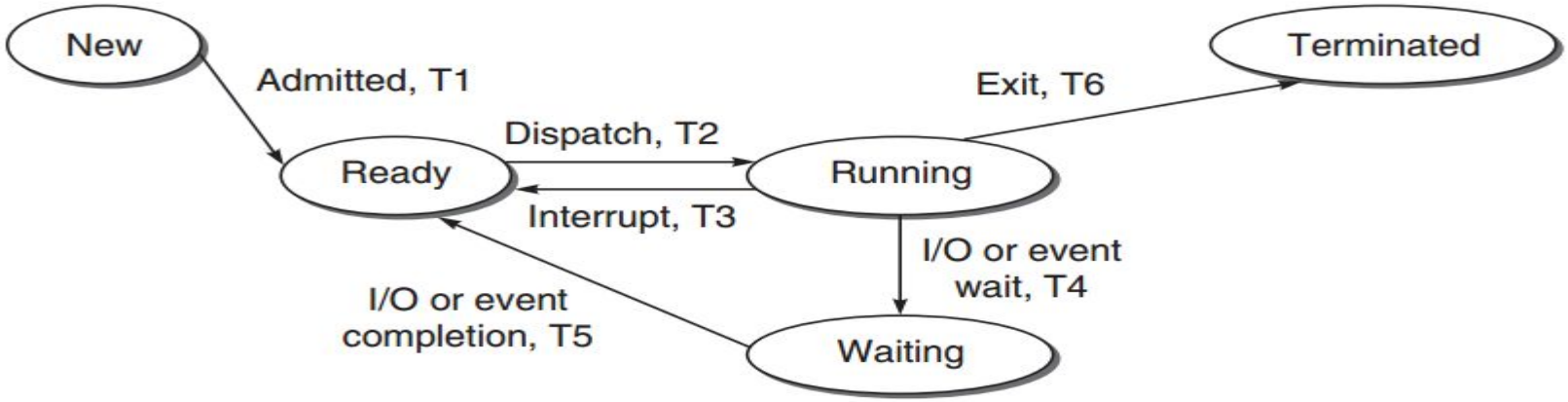
State Graph


New
Admitted, T1
Terminated
Exit, T6
Dispatch, T2
Ready
Running
Interrupt, T3
I/O or event
wait, T4
I/O or event
completion, T5
Waiting

The Resulting Output From A State

- 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



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

- Identify the states
- Prepare state transition diagram after understanding transitions between states
- Convert the state graph into the state table as discussed earlier
- Analyze the state table for its completeness
- Create the corresponding test cases from the state table

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
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

Decision Table-based Testing

- Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. These consider each input separately. There may be some critical behavior to be tested when some combinations of input conditions are considered.
- 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. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

Decision Table Structure

		ENTRY				
		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		

Test Case Design Using Decision Table

- 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.

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.

Example

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		

Example

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		

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

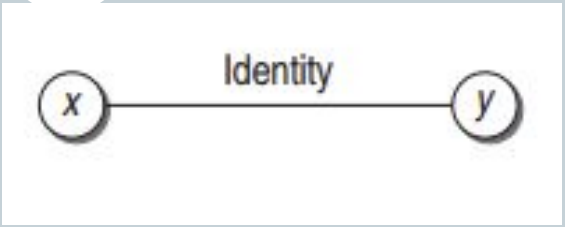
Cause-effect Graphing Based Testing

- The Cause and Effect Graph is a dynamic test case writing technique. Here causes are the input conditions and effects are the results of those input conditions.

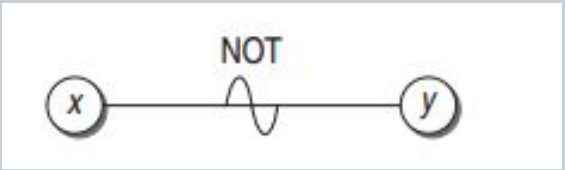
- Cause-Effect Graph is a technique that starts with a set of requirements and determines the minimum possible test cases for maximum test coverage which reduces test execution time and cost. The goal is to reduce the total number of test cases, still achieving the desired application quality by covering the necessary test cases for maximum coverage.

Basic Notation for Cause-Effect Graph

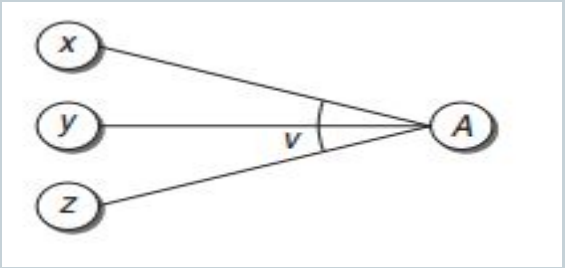
Identity



Not

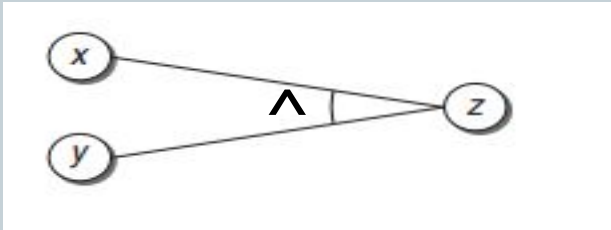


OR

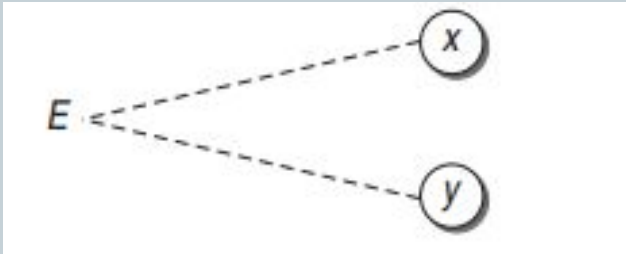


Basic Notation for Cause-Effect Graph

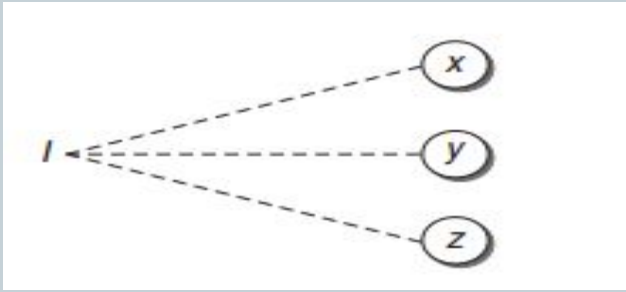
AND



Exclusive

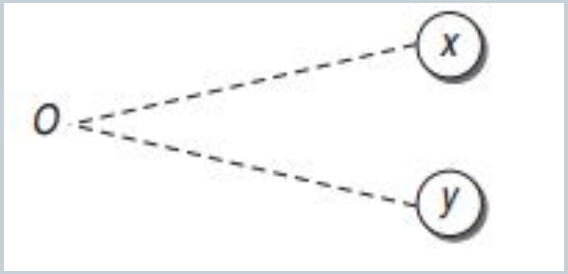


Inclusive

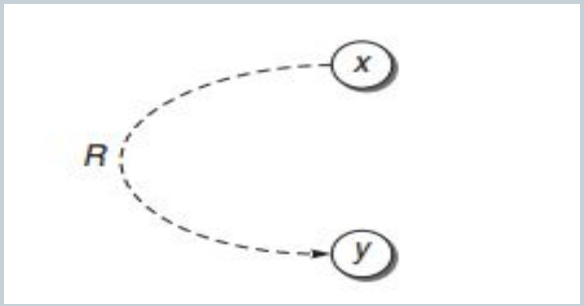


Basic Notation for Cause-Effect Graph

One and Only One



Requires



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.

Identify The Causes and Effects

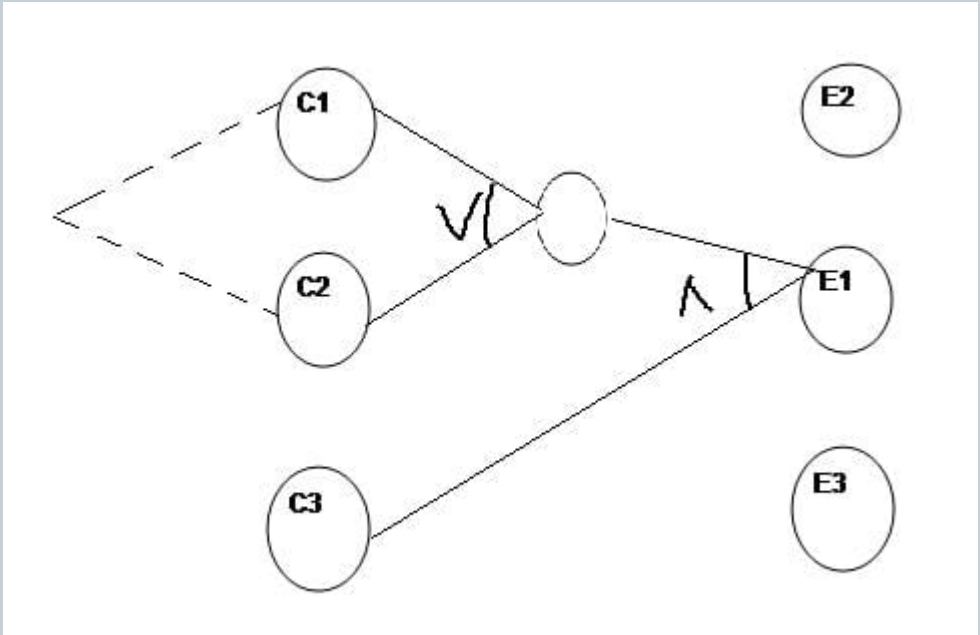
- **The Causes of this situation are:**
  C1 – First character is A
  C2 – First character is B
  C3 – the Second character is a digit
- **The Effects (results) for this situation are:**
  E1 – Update the file
  E2 – Print message “X”
  E3 – Print message “Y”

Nodes for Causes And Effects

Understand The Constraints for E1

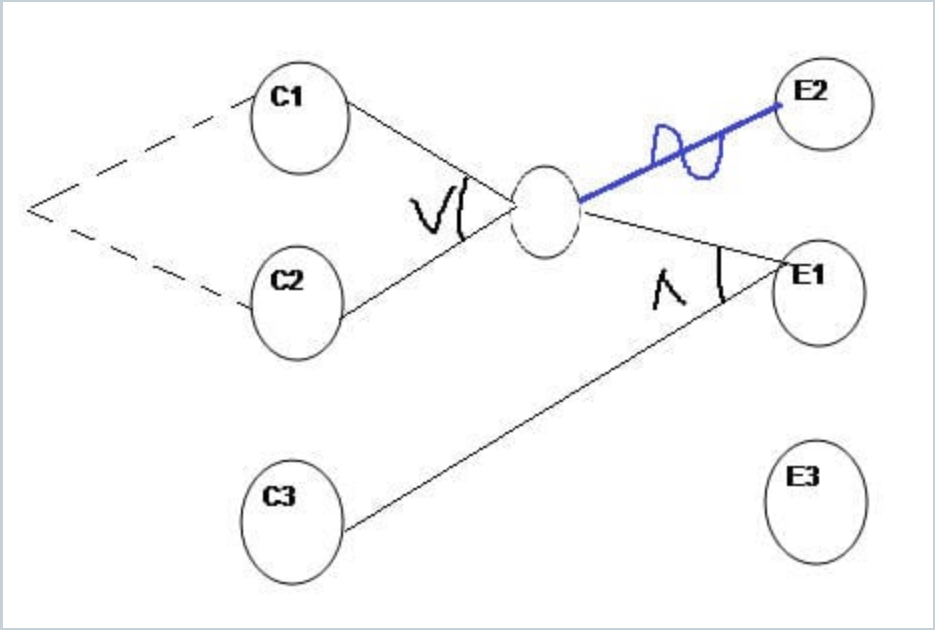
- Effect E1 is for updating the file. The file is updated when
  - The first character is “A” and the second character is a digit
  - The first character is “B” and the second character is a digit
  - The first character can either be “A” or “B” and cannot be both.
- Now let’s put these 3 points in symbolic 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.

Represent The Constraints in The Graph



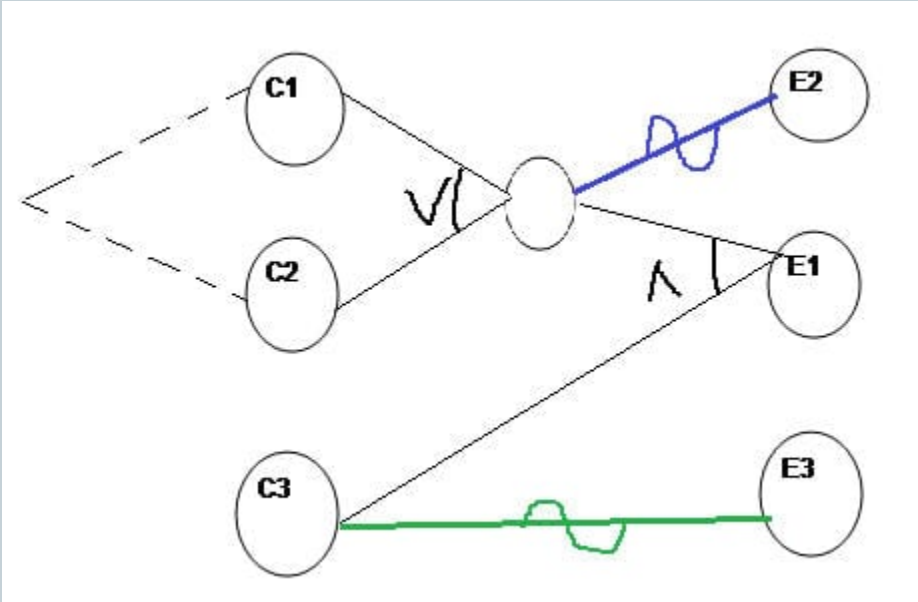
Draw Edges for E2

- 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)



Draw Edges for E3

- 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)



Decision Table

Actions
C1
C2
C3
E1
E2
E3

Actions	
C1	
C2	
C3	
E1	1
E2	
E3	

Decision Table

Now for E1 to be “1” (true), we have the below two conditions –
C1 AND C3 will be true
C2 AND C3 will be true

Actions
C1
C2
C3
E1
E2
E3

Actions	
C1	
C2	
C3	
E1	1
E2	
E3	

Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

Decision Table

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Thank You

END OF CHAPTER