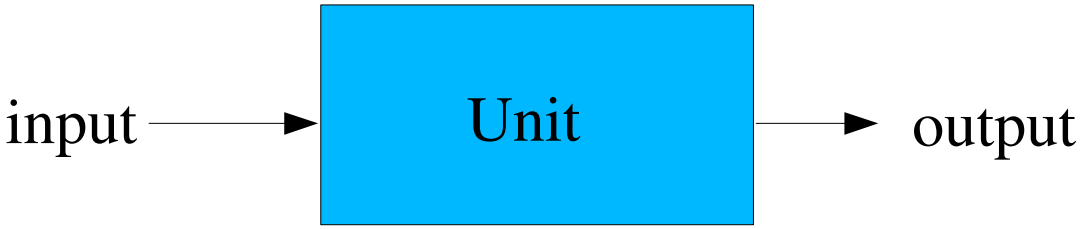
Black box testing

What is Black-Box testing ?

- Testing without having an insight into the details of underlying code



- Testing based on *specification*

What is Black-Box testing ?

- Advantages ?

- Disadvantages ?

What is Black-Box testing ?

- Approaches:
  - Error guessing
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Category Partitioning
  - Decision tables
  - Cause-Effect Graphing

Error Guessing

- Ad-hoc approach based on experience
- Approach:
  1. Make list of possible errors or error-prone situations
     - ➔ error model
  2. Design test-cases to cover error model
- Develop and maintain your own error models

Error Guessing

- Example sorting array function
- Error model:
  - empty array
  - already sorted array
  - reverse-sorted array
  - large unsorted array
  - ...
- Generate test cases for these situations

Equivalence Partitioning

- Partition input domain in equivalence classes according to specification
- Ideally ECs should be
  - such that each input is a class
  - disjointed
  - elements of same class mapped to output similarly
- Difficult in practice to find *perfect* ECs
  - use of heuristics

Heuristics for Identifying Equivalence Classes

For each external input and output:

1. if specifies range of valid values, define
   - 1 valid EC (within the range)
   - 2 invalid EC (one outside each end of the range)
2. if specifies a number (N) of valid values, define
   - 1 valid EC and
   - 2 invalid ECs (none and more than N)
3. if specifies a set of valid values, define
   - 1 valid EC (within set) and
   - 1 invalid EC (outside set)

Heuristics for Identifying Equivalence Classes

4. if specifies a *must be* situation, define
   - 1 valid EC (satisfy must) and
   - 1 invalid EC (don't satisfy must)
5. if there is a reason to believe that elements in an EC are not handled in an identical manner by the program,
   - subdivide EC into smaller Ecs.

Equivalence Class Partitioning

- Consider creating an equivalence partition that handle the default, empty, blank, null, zero, or none conditions.

Equivalence Classes Partitioning - Triangle Example

Function: *determineTriangle(a, b, c)*

Specification

- input three integers (sides of a triangle: a, b, c)
  - each side must be a positive number not greater than 1000
- output type of the triangle:
  - Equilateral if $a = b = c$
  - Isosceles if 2 pairs of sides are equals
  - Scalene if no pair of sides is equal or
  - NotATriangle if $a >= b + c$, $b >= a + c$, *or* $c >= a + b$

Equivalence Classes Partitioning - Triangle Example

Input Condition	Valid EC	Invalid EC
a	0 < a <= 1000 (1)	a <= 0 (8) a > 1000 (9) a not integer (10)
b	0 < b <= 1000 (2)	b <= 0 (11) b > 1000 (12) b not integer (13)
c	0 < c <= 1000 (3)	c <= 0 (14) c > 1000 (15) c not integer (16)

Output Condition	Valid EC	Invalid EC
returned value	Equilateral (4) Isosceles (5) Scalene (6) NotATriangle (7)	Anything else (17)

Equivalence Classes Partitioning – Identifying test cases

- Based on *coverage goal*
  - execute each Equivalence Class at least once
- Approaches for creating tests for coverage
  - One-to-One: one test case per EC
  - Minimized: as many ECs covered as possible per test cases
  - Myer's Selection Approach
  - Combinatorial: e.g. pair-wise

Equivalence Classes Partitioning – Identifying test cases

Myer's Selection Approach

1. Until all valid ECs have been covered by test cases,
   - write a new test case that cover as many of the uncovered valid ECs as possible (minimized)
2. Until all invalid ECs have been covered by test cases,
   - write a test case that covers *one, and only one* of the uncovered invalid ECs (one-to-one)

Equivalence Classes Partitioning - Triangle Example

- Test cases for triangle – Myer's selection

EC	a	b	c	Expected Output
1, 2, 3, 4	120	120	120	Equilateral
1.2, 3, 5	6	6	7	Isosceles
1, 2, 3, 6	925	312	704	Scalene
1, 2, 3, 7	8	2	3	Not a triangle
8	-10	15	6	Error
9	5000	21	19	Error
10	xhello	14	22	Error
11	325	-200	11	Error
12	12	1010	10	Error
13	8	!	18	Error
14	120	120	-120	Error
15	85	70	10000	Error
16	6	6		Error
17	IMPOSSIBLE			

Equivalence Partitioning

- Advantages
  - Small number of test cases needed
  - Probability of uncovering defects with the selected test cases higher than that with a randomly chosen test suite of the same size
- Limitations
  - Specification doesn't always define expected output for *invalid* test-cases
  - Strongly typed languages eliminate the need for the consideration of some invalid inputs
  - Myer's test selection approach weak when input variables *are not independent*

Example - Nextdate

- Input: *month*, *day*, *year* representing a *date*
  - *1 <= month <= 12*
  - *1 <= day <= 31*
  - *1812 <= year <= 2100*
- Output: date of the day after the input date
  - Must consider leap years
  - Leap year if divisible by 4, and not a century year
  - Century year is leap years if multiple of 400

Example - Nextdate

- Equivalence classes

Input Condition	Valid Ecs	Invalid Ecs
month	1 <= month <= 12	month < 1 month > 12
day	1 <= day <= 31	day < 1 day > 31
year	1812 <= year <= 2100	year < 1812 Year > 2100

- Refinement can be done based on how input is treated
  - Day incremented by 1 unless it's last day of a month
    - Depend on month (30, 31, February)
  - At end of month next day is 1, month is incremented
  - At end of year, both days and months are reset to 1, and year incremented
  - Problem of leap year

Example - Nextdate

- Decomposition of valid ECs

Input Condition	Valid Ecs
month	30 days (1)
	31 days except December (2)
	December (3)
	February (4)
day	1 <= day <= 27 (5)
	day = 28 (6)
	day = 29 (7)
	day = 30 (8)
	day = 31 (9)
year	1812 <= year <= 2100 and century and year mod 400 <> 0 (10)
	1812 <= year <= 2100 and century and year mod 400 = 0 (11)
	1812 <= year <= 2100 and year mod 4 <> 0 (12)
	1812 <= year <= 2100 and not century and year mod 4 = 0 (13)

Example - Nextdate

- Test cases obtained with Myer's approach

ID	Ecs	month	day	year	output
1	1, 5, 10	6	14	1900	1900-06-15
2	2, 6, 11	1	28	2000	2000-01-29
3	3, 7, 12	12	29	2001	2001-12-30
4	4, 8, 13	2	30	2012	Error
5	9	6	31	1977	Error

- Better coverage obtained by selecting test cases according Cartesian product of ECs

  ➔ $4 \times 5 \times 4 = 80$ test cases.

Example - Nextdate

- Test cases by Cartesian product

ID	Ecs	month	day	year	output
1	1, 5, 10	6	14	1900	1900-06-15
2	1, 5, 11	6	14	2000	2000-06-15
3	1, 5, 12	6	14	1913	1913-06-15
4	1, 5, 13	6	14	1912	1900-06-15
5	1, 6, 10	6	28	1900	1900-06-29
6	1, 6, 11	6	28	2000	2000-06-29
7	1, 6, 12	6	28	1913	1913-06-29
8	1, 6, 13	6	28	1912	1912-06-29
9	1, 7, 10	6	29	1900	1900-06-30
10	1, 7, 11	6	29	2000	2000-06-30
11	1, 7, 12	6	29	1913	1913-06-30
12	1, 7, 13	6	29	1912	1912-06-30
...	...	...	...	...	...

Equivalence Classes Partitioning

- *Brute-force* of defining a test case for every combination of the inputs ECs
  - provides good coverage, but
  - impractical when number of inputs and associated classes is large
  - possibility for redundancies

Equivalence Classes Partitioning

- Lower levels of combination may be used to obtain fewer test-cases
- *t-way* coverage
  - set of tests covers all combinations of *t* elements
    - 1<= t <= k (k total number of parameters)
    - example: t = 2 for pairwise coverage
  - *Covering array* - mathematical object that covers all t-way combinations of parameter values at least once

Covering Arrays

- Example suppose 3 parameters a, b, c each with 2 classes of values

Full Table

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Covering Array for 2-way coverage

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Example - Nextdate

Covering Array for 2-way (pairwise) coverage

	month	day	year
1	EC1	EC5	EC11
2	EC2	EC5	EC12
3	EC3	EC5	EC13
4	EC4	EC5	EC10
5	EC1	EC6	EC12
6	EC2	EC6	EC13
7	EC3	EC6	EC10
8	EC4	EC6	EC11
9	EC1	EC7	EC13
10	EC2	EC7	EC10
11	EC3	EC7	EC11
12	EC4	EC7	EC12
13	EC1	EC8	EC10
14	EC2	EC8	EC11
15	EC3	EC8	EC12
16	EC4	EC8	EC13
17	EC1	EC9	EC10
18	EC2	EC9	EC11
19	EC3	EC9	EC12
20	EC4	EC9	EC13

Boundary Value Analysis

- Errors tend to occur near extreme values (*boundaries*)

- BVA enhances Equivalence Partitioning by
  - selecting elements just on, and just beyond the borders of each EC

Boundary Conditions

- Situation at the edge of the planned operational limits of the software
- Data types with boundary conditions
  - Numeric, Character, Position, Quantity, Speed, Location, Size
- Boundary condition characteristics
  - First/Last, Start/Finish, Min/Max, Over/Under, Empty/Full, Slowest/Fastest, Largest/Smallest, Next-To/Farthest-From, Shortest/Longest, Soonest/Latest, Highest/Lowest

Boundary Value Analysis - Guidelines

- For each boundary condition
  - include boundary value in at least one valid test case
  - include value just beyond boundary in at least one invalid test case

- Use same guidelines for each output condition
  - include test cases with input such that output at boundaries are produced (if possible)

Boundary Value Analysis

- Sub-Boundary Conditions
  - internal to software (not defined in specification)
  - examples:
    - Powers-of-two: bit, nibble, byte, word, kilo, mega, giga, tera
    - ASCII table

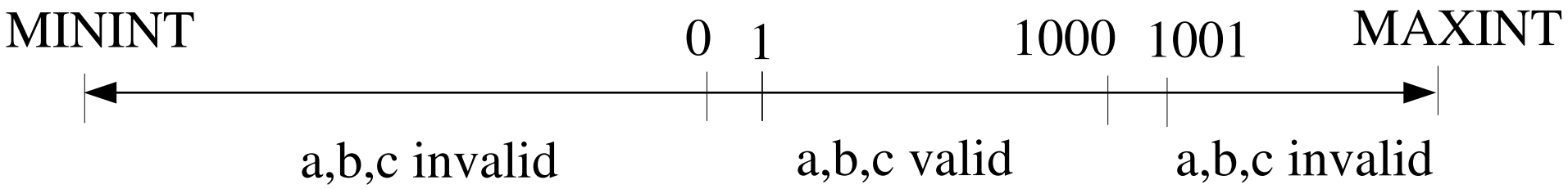
Boundary Value Analysis

	Character	ASCII Code
	/	47
lower bound	0	48
	1	49
	2	50
	3	51
	4	52
	5	53
	6	54
	7	55
	8	56
upper bound	9	57
	:	58
	A	65
	a	97

If numbers are expected (conversion from ascii), '/' and ':' are possible boundary values

Boundary Value Analysis

Boundary Conditions for Triangle



Category Partitioning

- Systematic, specification based methodology that uses an informal functional specification to produce formal test specification
- Consists in:
    - identification of *categories*,
    - partitioning each category in *choices*,
    - creating *test frames* as selection of choices and
    - creating *test cases* from *test frames*

Category Partitioning - Steps

- Decompose the functional specification into functional units if needed
    - to be tested independently
- Examine each functional unit
    - Identify Parameters
        - Explicit input to the functional unit
    - Identify Environmental conditions
        - Characteristics of the system's state

Category Partitioning - Steps

- Find categories for each parameter & environmental condition
    - Major property or characteristic of a parameter/condition
- Find choices for each category
    - Distinct partition of a category

Category Partitioning - Steps

- Determine constraints among the choices.
    - how the choices interact, how the occurrence of one choice can affect the existence of another, and what special restrictions might affect any choice.

- Create test frames
    - set of choices, one from each category
- Create test cases
    - test frame with specific values for each choices.

Category Partitioning - Example

Command: find

Syntax: find <pattern> <file>

Function: The find command is used to locate one or more instance of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file .To include a blank in the pattern, the entire pattern must be enclosed in quotes (“).To include quotation mark in the pattern, two quotes in a row (“ “) must be used.

Examples:

find john myfile

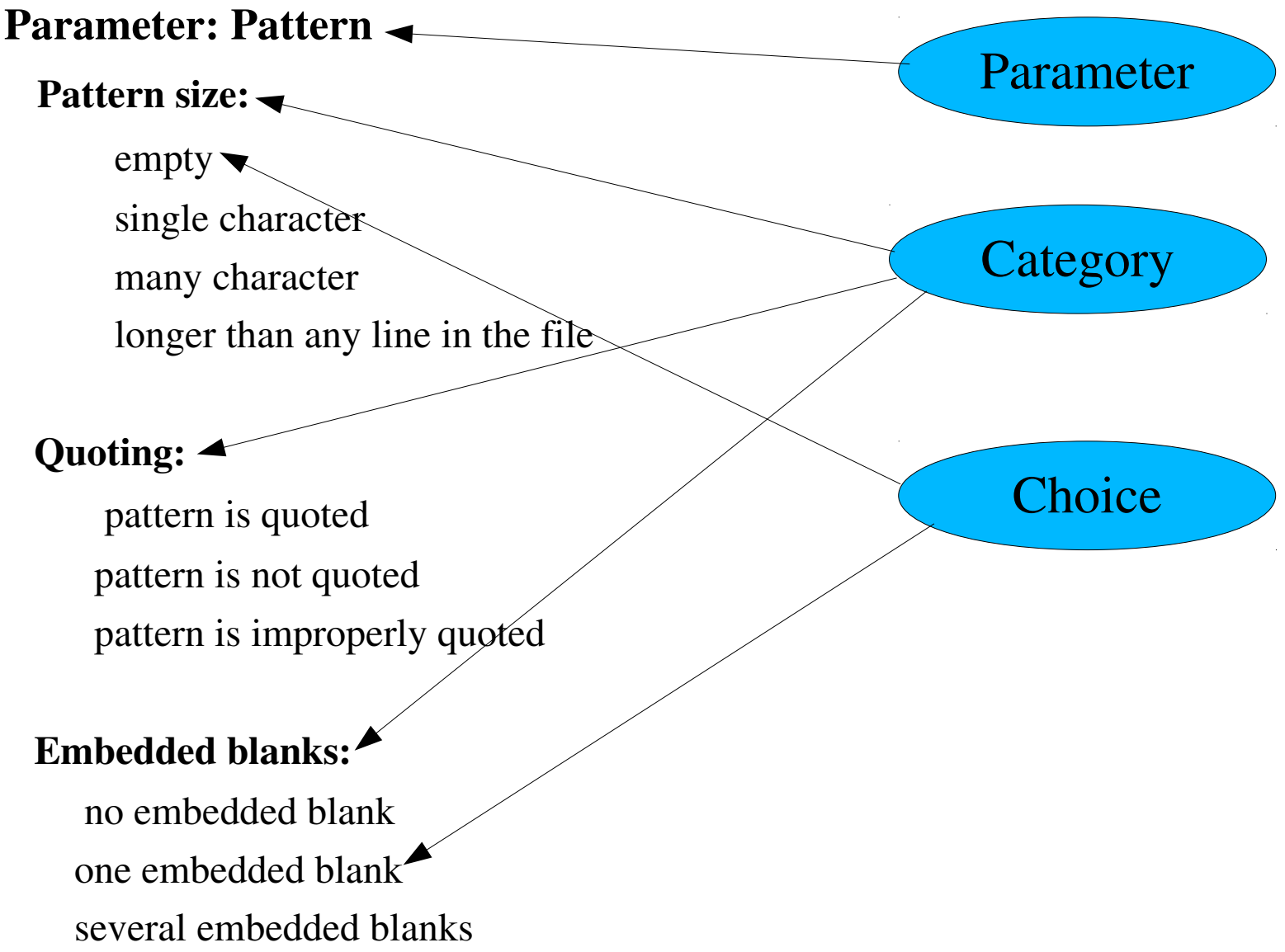
displays lines in the file **myfile** which contain **john**

find “john smith” myfile

displays lines in the file **myfile** which contain **john smith**

find “john”” smith” myfile

displays lines in the file **myfile** which contain **john” smith**


Parameter: Pattern
Pattern size:
empty
single character
many character
longer than any line in the file
Quoting:
pattern is quoted
pattern is not quoted
pattern is improperly quoted
Embedded blanks:
no embedded blank
one embedded blank
several embedded blanks
Parameter
Category
Choice

Embedded quotes:

no embedded quotes

one embedded quotes

several embedded quotes

Parameter: File

File name:

good file name

no file with this name

omitted

Environments:

Number of occurrence of pattern in file:

none

exactly one

more than one

Pattern occurrences on target line:

one

more than one

Test Frame - Example:

Pattern size : many characters

Quoting : pattern is quoted

Embedded blanks : several embedded blanks

Embedded quotes : no embedded quote

File name : good file name

Number of occurrence of pattern in file : none

Pattern occurrence on target line : one

Category Partionning - Constraints

- To eliminate contradictory/unfeasible frames
- Properties
  - assigned to choices, and tested for by other choices.
  - Format: [property A, B, ...]
    - A and B are property names
    - E.g., [property Empty]

Category Partionning - Constraints

- Selector expression
  - conjunction of properties assigned to other choices
  - [if A]
    - E.g., [if Empty]

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[property NonEmpty]

Quoting:

pattern is quoted	[property Quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[if NonEmpty]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

no embedded quotes	[if NonEmpty]
one embedded quotes	[if NonEmpty]
several embedded quotes	[if NonEmpty]

File name:

good file name

no file with this name

omitted

Environments:

Number of occurrence of pattern in file:

none	[if NonEmpty]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

one	[if Match]
more than one	[if Match]

Category Partionning – error annotation

- To limit redundant frames when some parameter or environment condition causes an error
- Annotation [error]
  - added to a choice that produces an error regardless of other choices
  - only one frame is created with the annotated choice
  - not systematic combination with choices in other categories to create test frames

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[error]

Quoting:

pattern is quoted	[property quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[error]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

no embedded quotes	[if NonEmpty]
one embedded quotes	[if NonEmpty]
several embedded quotes	[if NonEmpty]

File name:

good file name	
no file with this name	[error]
omitted	[error]

Environments:

Number of occurrence of pattern in file:

none	[if NonEmpty]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

one	[if Match]
more than one	[if Match]

Category Partionning – single annotation

- To describe special, unusual, or redundant conditions that do not have to be combined with all possible choices.
- Annotation [single]
  - judgment by the tester that the marked choice can be adequately tested with only one test case
  - only one frame is created with the annotated choice
  - not systematic combination with choices in other categories to create test frames

Embedded quotes:

no embedded quotes	[if NonEmpty]
one embedded quotes	[if NonEmpty]
several embedded quotes	[if NonEmpty] [single]

File name:

good file name	
no file with this name	[error]
omitted	[error]

Environments:

Number of occurrence of pattern in file:

none	[if NonEmpty] [single]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

one	[if Match]
more than one	[if Match] [single]

Decision Table

- Ideal for situations where:
  - combinations of actions taken under varying set of conditions
  - conditions depends on input variables
  - response produced doesn't depend on the order in which input variables are set or evaluated, and
  - response produced doesn't depend on prior input or output

Decision Table - Format

Conditions	Combination of conditions (variants)
Actions	Selected actions

Decision Table - Development

1. Identify decision variables and conditions
2. Identify resultant actions to be selected or controlled
3. Identify which action should be produced in response to particular combinations of actions

Example

- Suppose the following insurance renewal rules
  - 0 claims, age<=25: raise by $50
  - 0 claims, age>25: raise by $25
  - 1 claim, age<=25: raise by $100, send letter
  - 1 claim, age>25: raise by $50
  - 2, 3 or 4 claims, age<=25: raise by $400, send letter
  - 2, 3 or 4 claims, age>25: raise by $200, send letter
  - more than 4 reclamations: cancel policy

Decision Table – Don't Care condition

- Don't Care condition
  - May be *true* or *false* without changing the action
  - Simplifies the decision table
  - Corresponds to different implementation cases:
    - inputs are necessary but have no effect for the variant
    - inputs may be omitted but have no effect if supplied
    - **Type-Safe exclusion**
      - several conditions are defined for a non binary decision variable, which can hold only one value at a time

Decision Table – Can't Happen & Don't know conditions

Attention not to confuse a *Don't Care* condition with

- *Can't Happen* Assumption - reflects assumption that
  - some inputs are mutually exclusive,
  - some inputs can't be produced by the environment, or
  - implementation is structured so as to prevent evaluation
- *Don't Know* Situation – reflects an incomplete model

- Usually indication of mis-specification
  - tests needed to exercise these undefined cases

Decision Table – Triangle Example

- **Decision variables**: sides a, b, c
- **Conditions**
  - a < b+c
  - b < a+c
  - c < a+b
  - a=b
  - a=c
  - b=c
  - a > 1000
  - b > 1000
  - c > 1000
- **Actions**
  - finding it is not a triangle
  - finding it is a scalene triangle
  - finding it is an isosceles triangle
  - finding it is an equilateral triangle
  - finding it is an error
  - finding variant impossible

Decision Table – Triangle Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
c1: a<b+c	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c2: b<a+c	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c3: c<a+b	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c4: a=b	-	-	-	Y	Y	Y	Y	N	N	N	N	-	-	-
c5: a=c	-	-	-	Y	Y	N	N	Y	Y	N	N	-	-	-
c6: b=c	-	-	-	Y	N	Y	N	Y	N	Y	N	-	-	-
c7: a>1000	N	N	N	N	N	N	N	N	N	N	N	Y	-	-
c8: b>1000	N	N	N	N	N	N	N	N	N	N	N	-	Y	-
c9: c>1000	N	N	N	N	N	N	N	N	N	N	N	-	-	Y
a1: not a triangle	X	X	X											
a2: Scalene											X			
a3: Isosceles							X		X	X				
a4: Equilateral				X										
a5: impossible					X	X		X						
a6: error												X	X	X

- **Attention** for ***inconsistencies***

Decision Table – Test generation Triangle example

- One test per variant

Variant	a	b	b	Expected output
1	4	1	2	Not a Triangle
2	1	4	2	Not a Triangle
3	1	2	4	Not a Triangle
4	5	5	5	Equilateral
5				Impossible
6				Impossible
7	2	2	3	Isosceles
8				Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene
12	21	6	12	Error
13	6	45	17	Error

Decision Table – Test generation strategies

- Each-Condition/All-Conditions
  - Heuristic to reduce number of variants
- Set of variants includes
  - For each variable, a variant such that the variable is made true with all other variables being false, and
    - One variant such that all variables are true (*and* logic), or
    - One variant such that all variables are false (*or* logic)

Decision Table – Each-Condition/All-Conditions

S = P *or* Q *or* R

P	Q	R	S (action)
F	F	T	x
F	T	F	x
T	F	F	x
F	F	F	

Decision Table – Each-Condition/All-Conditions

S = P *and* Q *and* R

P	Q	R	S (action)
F	F	T	
F	T	F	
T	F	F	
T	T	T	x

Decision Table – Each-Condition/All-Conditions

Z = (A *and* B *and* (*not* C)) *or* (A *and* D)

Develop variants for (A *and* B *and* (*not* C)) and variants for (A *and* D)

A	B	C	D	Z (action)
T	F	T	-	
F	T	T	-	
F	F	F	-	
T	T	F	-	x
T	-	-	F	
F	-	-	T	
T	-	-	T	x

- Values must be assigned to Don't Care variables
  - randomly or
  - by suspicion

Decision Table – Each-Condition/All-Conditions

Z = (A *and* B *and* (*not* C)) *or* (A *and* D)

A	B	C	D	Z (action)
T	F	T	T	
F	T	T	F	
F	F	F	F	
T	T	F	F	x
T	F	F	F	
F	T	T	T	
T	T	T	T	x

- Example of variants used for test suite generation

Cause-effect graphing

- Systematic way to aid selecting a high-yield set of test cases
  - to identify/analyze relationships in a decision table
  - generate boolean formula
- Nodes:
  - Cause - distinct instance of input condition or EC
  - Effect - observable result of change in system state
  - Intermediate node - combination of causes representing an expression on input conditions

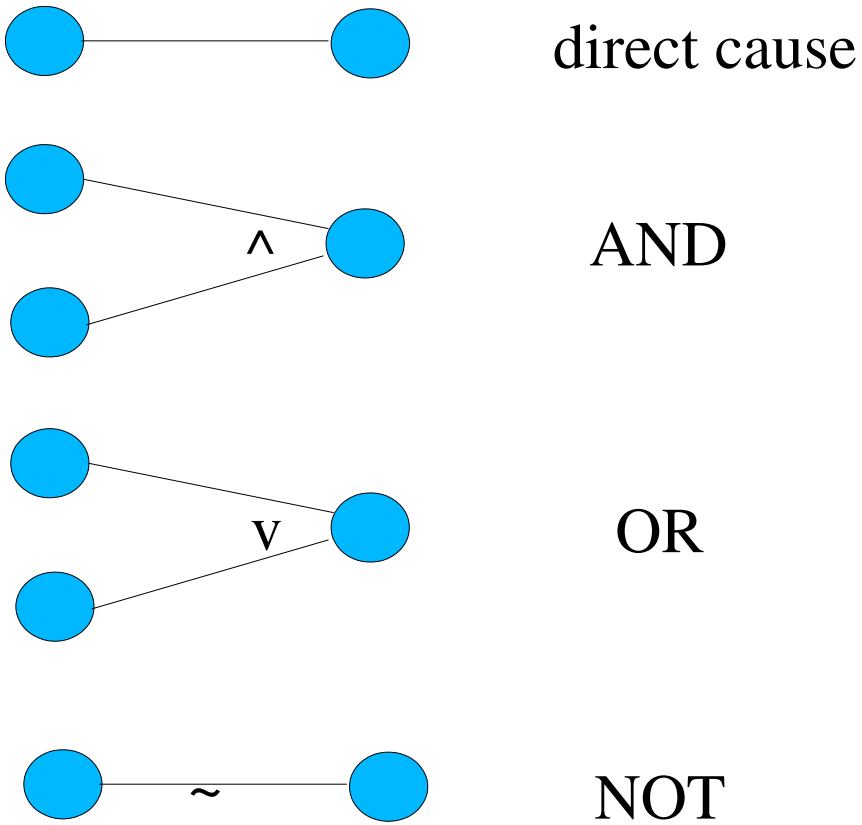
Cause-effect graphing - layout

Causes Intermediate nodes Effects

Valid links from

- cause to intermediate node
- cause to effects
- intermediate node to intermediate node
- intermediate node to effect

Cause-effect graphing – type of links



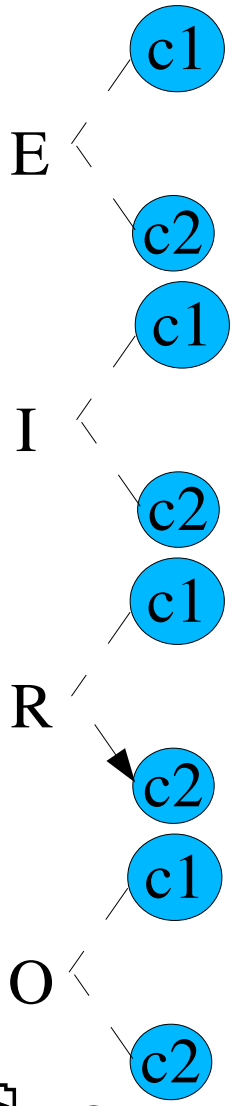
Cause-effect graphing - example

A file update depends on the value in two fields. The value in field 1 must be an ``A'' or ``B''. The value in field 2 must be a digit. In this situation the file update is made. If the value in field 1 is incorrect, error message X12 is issued. If the value in field 2 is incorrect, error message X13 is issued.

Cause-effect graphing - example

- Causes
  - c1: character in column 1 is "A"
  - c2: character in column 1 is "B"
  - c3: character in column 2 is a digit
- Effects
  - e1: update made
  - e2: message X12 issued
  - e3: message X13 issued

Cause-effect graphing – constraints between causes



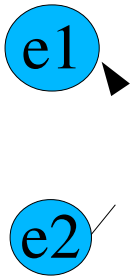
Exclusive – c1 and c2 can not be true simultaneously

Inclusive – one of c1 and c2 must be true (can't be both false)

Require – c2 must be true for c1 to be true

One and only One – one and only one of c1 and c2 can be true at a given time

Cause-effect graphing – constraints between effects



Mask – e1 forced to false if e2

Test generation from Cause-Effect graphs

- Divide the specification into *workable* pieces.
- Define separate graphs for each piece.
  1. Identify Causes and Effects
  2. Deduce Logical Relationships and Constraints – represent as graph
  3. Draw a decision table
  4. Convert each variant in the decision table into a test case.

Test generation from Cause-Effect graphs

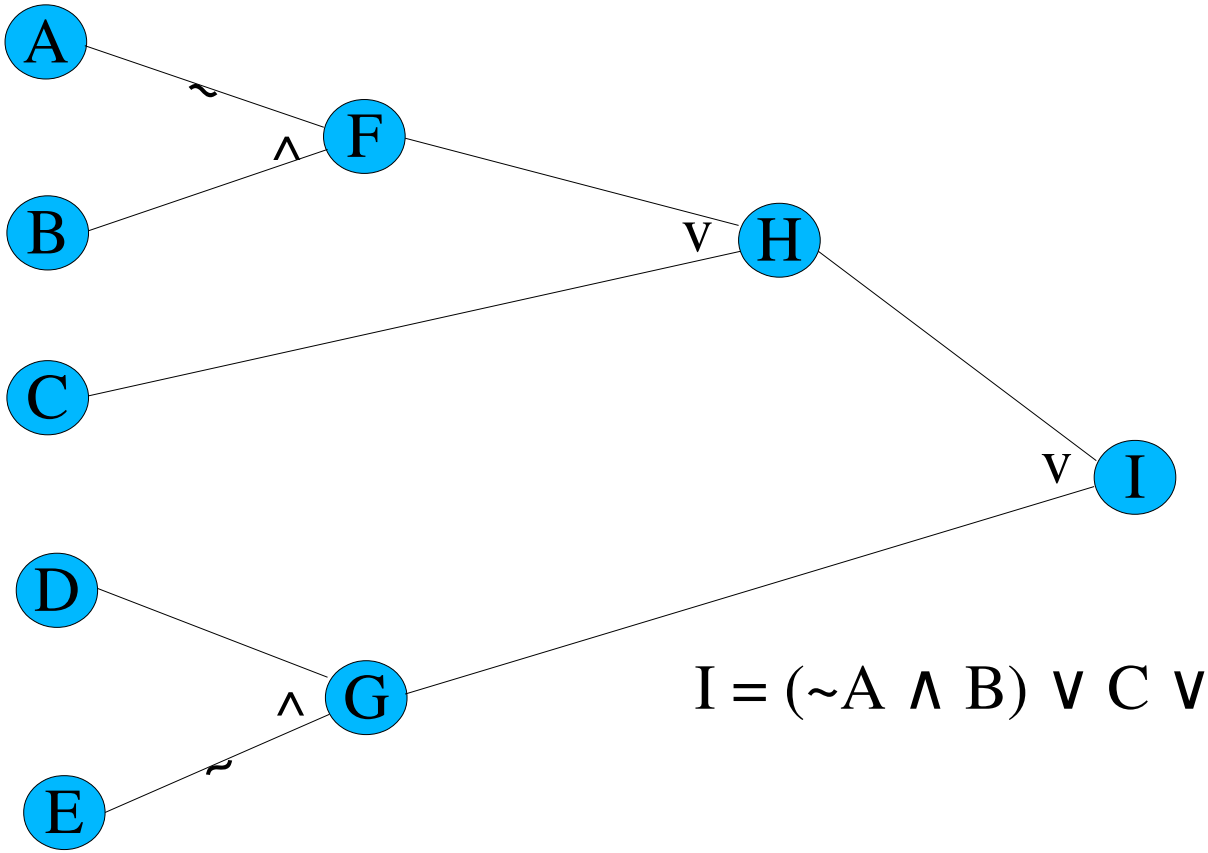
- To find the unique combination of conditions
- Work with a single effect (output) at a time
  1. Set effect to *true* (1) state
  2. Look backward for all combination of inputs which will force effect to the true state (constraints limit number of combination)
  3. Create a column in decision table for each combination
  4. Determine states of all other effects for each combination

Boolean Formula Generation from Cause-Effect graphs

- Working from effects toward causes

1. Transcribe node-to-node formulas from the graph
   - write formula for each effect and its predecessors
   - For each intermediate node
     - write formula for intermediate node and its predecessors
2. Derive the complete Boolean formula
   - replace intermediate variables by substitution until the effect formula contains only cause variables
   - factor and rewrite formula into sum-of-products form

Boolean Formula Generation from Cause-Effect graphs



- I = H ∨ G
- H=F ∨ C
- F=~A ∧ B
- G=D ∧ ~E

$$I = (\sim A \wedge B) \vee C \vee (D \wedge \sim E)$$